Capstone Project Phase B

AutoScope – Application for ear infection detection

24-2-D-24

Authors:
Alon Ternerider
Nadav Goldin

Supervisor: Dr. Natali Levi

GitHub: https://github.com/ndvp39/autoscope

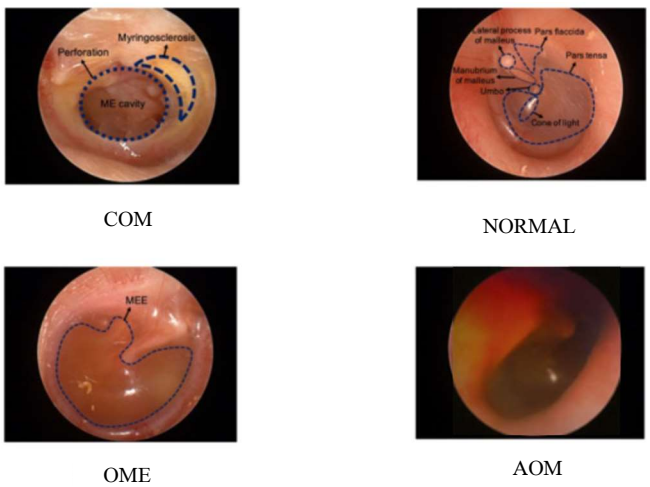
Table Of Contents

Abstract

With the rising demand for accessible and accurate diagnostic tools, this project introduces *AutoScope*, a desktop application designed for ear infection detection. The application integrates a USB otoscope with deep learning technologies to provide an innovative and user-friendly platform for analyzing ear conditions. Users can capture ear images directly through the application or upload existing images for analysis. A trained deep learning model, leveraging TensorFlow, processes the captured images to detect potential signs of infection. The *AutoScope* application incorporates Kivy for a cross-platform user interface and OpenCV for hardware communication, ensuring seamless integration with otoscope devices. Results are displayed in an intuitive desktop interface designed for ease of use. While the application offers a valuable preliminary diagnostic aid, it does not replace professional medical evaluations. Future iterations will explore enhancing model accuracy, expanding otoscope compatibility, and integrating additional diagnostic features to further assist healthcare providers and individuals in managing ear health.

Introduction

Ear infections (Otitis media) are a prevalent concern, especially among young children. These infections, if not detected and treated early, can lead to complications, prolonged discomfort, and the overuse of antibiotics. The excessive use of antibiotics contributes to the growing issue of antibiotic resistance, a significant public health threat. Timely detection and treatment of ear infections can reduce the need for systemic antibiotics, favoring localized treatments such as ear drops, which are less likely to contribute to resistance.
Otitis media (OM), refers to an illness usually found in children below the age of three years. In the U.S., the medical expenses of ear infections are predicted to be $ 3 to 5 billion annually. Three kinds of OM exist: chronic otitis media with effusion, acute otitis media (AOM), and otitis media with effusion (OME). Due to bacterial infection, AOM may occur in the middle of the ear, bringing about the build-up of fluid. OME leads to fluid build-up in the middle of the ear because of inflammation, which is much more severe than AOM.



COM NORMAL

OME AOM

Source: Automatic detection of tympanic membrane and middle ear infection from oto-endoscopic images via convolutional neural networks, 2020.
Intelligent Control Techniques for the Detection of Biomedical Ear Infection, 2022

Figure1: Sample images of different ear infection

Current solutions for diagnosing ear infections rely heavily on clinical examinations conducted by healthcare professionals using an otoscope. These examinations, though effective, are often limited by accessibility, requiring patients to visit clinics or hospitals. By the time the patient is examined, the ear infection can be worsened.
In some cases, diagnostic accuracy can be affected by the practitioner's experience and the subjective nature of visual assessments. There are also digital otoscopes available that can capture images of the ear canal, but they typically lack integrated diagnostic capabilities and require manual interpretation by a medical professional.
Our project aims to develop an innovative solution combining a digital otoscope with a desktop application that utilizes advanced image processing techniques to detect early signs of ear infections. The digital otoscope will capture high-quality images of the ear canal and eardrum, which will then be analyzed by our application. The app will leverage machine learning algorithms trained on a vast dataset of ear images to identify potential infections with high accuracy.
Our solution will provide a reliable, accessible, and user-friendly method for early detection of ear infections, potentially reduce the dependency on antibiotics by enabling prompt treatment with ear drops and the need for clinical visits.

The primary beneficiaries of our solution are young children and their parents. Children are particularly vulnerable to ear infections, and early detection can prevent severe discomfort and complications. Parents will benefit from the convenience of monitoring their child's ear health from home, reducing the frequency of doctor visits and the associated costs and time.
Healthcare providers will also find value in our solution. It can serve as a decision-support tool, assisting them in making accurate diagnoses more quickly and efficiently. This can lead to better patient outcomes and optimized use of medical resources.
Moreover, public health systems stand to gain from a reduction in antibiotic prescriptions. By minimizing unnecessary antibiotic use, our solution can contribute to the broader effort of combating antibiotic resistance, ultimately benefiting society.

2. Background

2.1 Existing Tools

2.1.1 Traditional Otoscope

The traditional otoscope is a handheld device extensively utilized by healthcare professionals to examine the ear canal and eardrum. It consists of a light source and a magnifying lens, enabling visual inspection of the ear's internal structures.
The effectiveness of a traditional otoscope is contingent upon the practitioner's expertise and experience. Additionally, the requirement for patients to visit a healthcare facility for examination limits accessibility and convenience. This tool also lacks any automated diagnostic features, relying solely on the clinician's interpretation.

2.1.2 Telemedicine Solutions

Telemedicine platforms (for example TytoCare[1]) leverage digital communication technologies to facilitate remote consultations between patients and healthcare professionals. These platforms often incorporate digital otoscopes to enable remote examination of the ear.
The efficacy of telemedicine solutions is dependent on the availability and promptness of healthcare professionals to interpret the received images. This can result in potential delays in diagnosis and treatment. Additionally, these platforms often do not incorporate automated diagnostic tools, thus relying on human expertise.

2.1.3 Mobile Applications For Digital Otoscope

Digital otoscopes (Oto by CellScope[2]) are advanced versions of traditional devices capable of capturing high-resolution images and videos of the ear canal and eardrum. These devices can integrate with mobile applications (HearScope[3]) designed to assist users in managing ear health by providing educational resources, symptom checkers, and enabling image capture for remote analysis.
Despite their ability to capture detailed images and offer some guidance, these tools generally lack sophisticated diagnostic capabilities. They still require manual interpretation by healthcare professionals or rely on basic algorithms, limiting their reliability and diagnostic accuracy compared to professional evaluations.

2.2 Early Detection

The early detection of ear infections is crucial for preventing a range of complications that can arise if these infections are not promptly treated. Complications from untreated ear infections include hearing loss, tympanic membrane perforation, and the spread of infection to nearby structures such as the mastoid bone, which can lead to mastoiditis[4]. By identifying infections at an early stage, appropriate treatments such as ear drops can be administered, which are less invasive and carry fewer side effects compared to systemic antibiotics[5].

Additionally, the overuse of systemic antibiotics for treating ear infections has contributed significantly to the global issue of antibiotic resistance. Antibiotic resistance occurs when bacteria evolve mechanisms to withstand the drugs used to treat infections, rendering standard treatments ineffective and leading to persistent infections[6]. By reducing the reliance on systemic antibiotics through early detection and localized treatment options, the progression of antibiotic resistance can be mitigated, preserving the efficacy of these critical medications for future use[7].

The importance of early diagnosis is further highlighted by studies that demonstrate improved patient outcomes when ear infections are treated promptly. Early

---

[1] TytoCare website.
[2] CellScope.
[3] HearScope.
[4] Ken Kitamura et al. "Clinical Practice Guidelines for the diagnosis and management of acute otitis media (AOM) in children in Japan".
[5] KATHRYN M. HARMES et al. "Otitis Media: Diagnosis and Treatment".
[6] STEVEN L. SOLOMON et al. "Antibiotic Resistance Threats in the United States: Stepping Back from the Brink".
[7] Benjamin D. Brooks, Amanda E. Brooks. "Therapeutic strategies to combat antibiotic resistance".

intervention has been shown to reduce the duration of symptoms, lower the incidence of recurrent infections, and decrease the overall burden on healthcare systems[8]. Moreover, timely treatment helps in avoiding unnecessary healthcare visits and reduces the economic strain associated with long-term complications and treatments[9].

2.3 Medical Image Analysis

A considerable body of scholarly work underscores the efficacy of machine learning algorithms in the realm of medical image analysis. Techniques such as convolutional neural networks (CNNs) have demonstrated significant accuracy in various image classification and diagnostic tasks[10]. For instance, recent studies elucidate the transformative potential of deep learning in healthcare, particularly emphasizing improvements in diagnostic precision[10]. Furthermore, comprehensive reviews of deep learning methodologies applied to medical imaging highlight their capacity to enhance diagnostic processes significantly[11].

2.4 Applications of Ear Images Processing

To ensure reliable spectral data classification, several key preprocessing steps must be executed on multimode image data. Initially, all RGB images, except for white light and fluorescence images, are converted to grayscale. This is followed by flat-field correction to normalize the images. Misalignments caused by hand movements during image acquisition intervals are corrected through image registration. For autofluorescence images, the Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm is applied to enhance contrast, and then image registration is performed again. The preprocessed images are then stacked into a multimode image cube consisting of nine grayscale spectral images and the red (R), green (G), and blue (B) channels of the autofluorescence image. The performance of various data combinations from this multimode image cube is compared to identify the best data set for classifying ear infections, specifically Otitis Media with Effusion (OME), Acute Otitis Media (AOM), and Chronic Otitis Media (COM)[12].

To classify ear images, various machine learning and conventional spectral classification algorithms are applied, including Multilayer Perceptron (MP), Random Forest (RF), Logistic Regression (LR), Decision Trees (DTs), Naive Bayes (NB), Spectral Angle Mapper (SAM), and Euclidean Distance (ED). These algorithms have been widely used in spectral imagery data analysis[13] [14]. For training and testing, ground truth data is established with medical experts identifying specific regions for each class (OME, AOM, and COM). Binary masks are created and applied to the image cube, followed by data standardization to account for varying intensity levels across different channels. The dataset is then split into 80% for training and 20% for testing.

MP is used to classify multimode images with an input layer of 12 nodes (representing nine spectral images and the R, G, and B channels of one

[8] Mayo Clinic Stuff. "Ear infection (middle ear)".
[9] Jerome O Klein. "The burden of otitis media".
[10] Daniele Ravì et al. "Deep Learning for Health Informatics".
[11] Dinggang Shen et al. "Deep Learning in Medical Image Analysis".
[12] Stephen M .Pizer et al. "Adaptive histogram equalization and its variations".
[13] Sewoong Kim et al. "Smartphone-based multispectral imaging and machine-learning based analysis for discrimination between seborrheic dermatitis and psoriasis on the scalp".
[14] Thiago C. Cavalcanti et al. "Smartphone-based spectral imaging otoscope: System development and preliminary study for evaluation of its potential as a mobile diagnostic tool".

autofluorescence image), a hidden layer optimized to ten nodes, and an output layer of three nodes (OME, AOM, and COM). The performance of the MP model is compared to other classifiers, with parameters tuned for optimal results. For instance, the RF classifier designed for multimode data uses 58 estimators, a maximum tree depth of 10, and specific minimum samples split and leaf parameters. Similar detailed configurations are used for DTs and other classifiers to achieve the highest classification accuracy[13] [15] [16].
Other machine learning classifiers, including LR and NB, are also tested on various data combinations. For example, the RF classifier for multimode data is fine-tuned with 24 estimators and specific tree depths and minimum samples parameters, while the DT classifier is optimized similarly. The criteria for split quality measure is entropy, ensuring precise classification. Each classifier's performance is rigorously evaluated to determine the most effective algorithm for each data type, thereby ensuring robust and accurate classification of OME, AOM, and COM from multimode images[13] [14].

2.5 Risks

Developing a mobile application like *AutoScope* comes with several technical and operational risks. One primary concern is ensuring compatibility with a wide range of USB otoscope devices. Since different manufacturers implement varying protocols for their devices, ensuring seamless integration across all supported desktop platforms (Windows, macOS, and Linux) may require extensive testing and customization. Additionally, maintaining consistent performance in real-time video streaming and image capturing can be challenging, especially on devices with limited processing power. These challenges could potentially impact the app's usability and user experience, necessitating significant development and optimization efforts.
Another significant risk lies in the accuracy and reliability of the deep learning model used for ear infection detection. Since the model is trained on specific datasets, there is a potential for bias or misclassification if the test data significantly differs from the training data. Misdiagnosis or false negatives could undermine user trust and lead to improper medical decisions. Furthermore, the security of sensitive user data, such as captured images and analysis results, is a critical concern. Any breach of data could result in legal and ethical consequences, especially when dealing with health-related information. Addressing these risks involves employing robust data protection measures, regular model updates with diverse datasets, and extensive validation to ensure both accuracy and user privacy.

[15] Thiago C. Cavalcanti et al. "Intelligent smartphone-based multimode imaging otoscope for the mobile diagnosis of otitis media".
[16] P.K Goel et al. "Classification of hyperspectral data by decision trees and artificial neural networks to identify weed stress and nitrogen status of corn".

3. Deep Learning Models

.

3.1 ResNet50

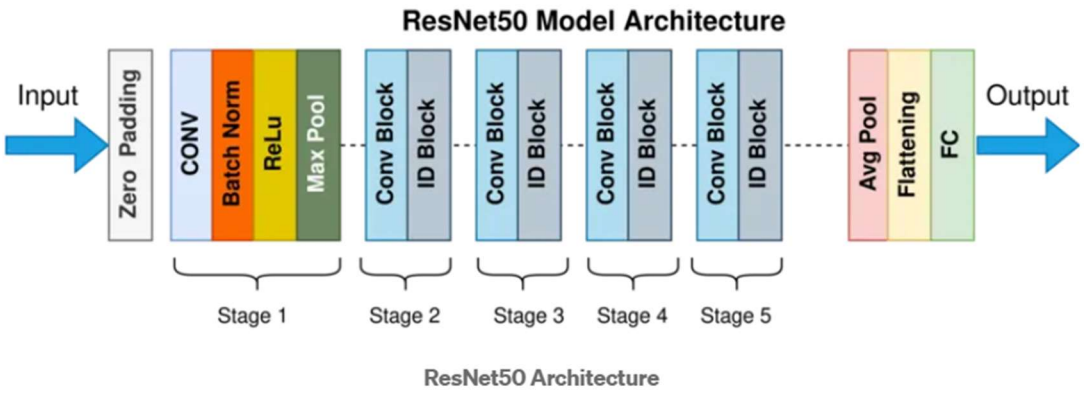


Figure 2: RestNer50 Model Architecture

ResNet50 is a deep convolutional neural network with 50 layers, designed to efficiently extract and process features from images. A key innovation in ResNet50 is the use of **skip connections**, which help mitigate the vanishing gradient problem, allowing the network to learn deep hierarchical features without losing accuracy.

In the context of ear infection detection, *ResNet50* processes eardrum images through multiple stages, including convolutional layers, identity blocks, and convolutional blocks, enabling the extraction of high-level features crucial for classification.
To adapt *ResNet50* for our classification task, we use a pretrained model with ImageNet weights as a feature extractor. The fully connected layers are removed while the convolutional base is retained.
The first layers remain frozen to preserve generic image representations, ensuring that the model retains fundamental patterns from natural images. To further adapt the network, the last 10 layers are fine-tuned, enabling the model to specialize in distinguishing between infected and normal eardrum images.

Preprocessing is a critical step in preparing the dataset for training. All images are resized to 224×224 pixels to match *ResNet50's* input size. Additionally, pixel values are normalized using the default *ResNet50* preprocessing function to improve learning efficiency. To enhance model generalization, data augmentation techniques are applied during training, including rotation to simulate different camera angles, brightness adjustment to improve robustness against lighting variations, and horizontal flipping to ensure the model learns invariant features across orientations.
Feature extraction is performed using the ResNet50 convolutional base with its classification layers removed.
The lower layers, responsible for capturing basic patterns like edges and textures, remain frozen to retain their pre-learned representations. Meanwhile, the last 10 layers are fine-tuned to adapt to infection-specific patterns, enhancing the model's classification accuracy.
To classify extracted features, a custom classification head is added. A Global Average Pooling (GAP) layer is used to convert feature maps into a single vector per image, reducing overfitting by minimizing the number of trainable parameters.

This is followed by a dense layer with 128 neurons and ReLU activation, which allows the model to learn deeper feature representations. To further prevent overfitting, a Dropout layer (40%) is applied, randomly deactivating neurons during training to improve generalization. Finally, a dense output layer with a single neuron and sigmoid activation produces a probability score between 0 and 1, where values closer to 0 indicate a normal eardrum, and values closer to 1 suggest an infection. For training, we use Stochastic Gradient Descent (SGD) with momentum (0.9) to ensure stable convergence, preventing sudden weight fluctuations while improving training speed. Since this is a binary classification task, Binary Cross-Entropy is used as the loss function. Given the common issue of class imbalance in medical datasets, class weights are computed and assigned, preventing the model from favoring the dominant class and ensuring better learning across both infected and normal cases. To further improve model performance and prevent overfitting, regularization techniques are incorporated. Early stopping is implemented to halt training when validation loss ceases to improve, preventing unnecessary training cycles. Additionally, learning rate reduction is used to fine-tune the model, decreasing the learning rate if validation loss stagnates, ensuring better optimization during later training epochs.

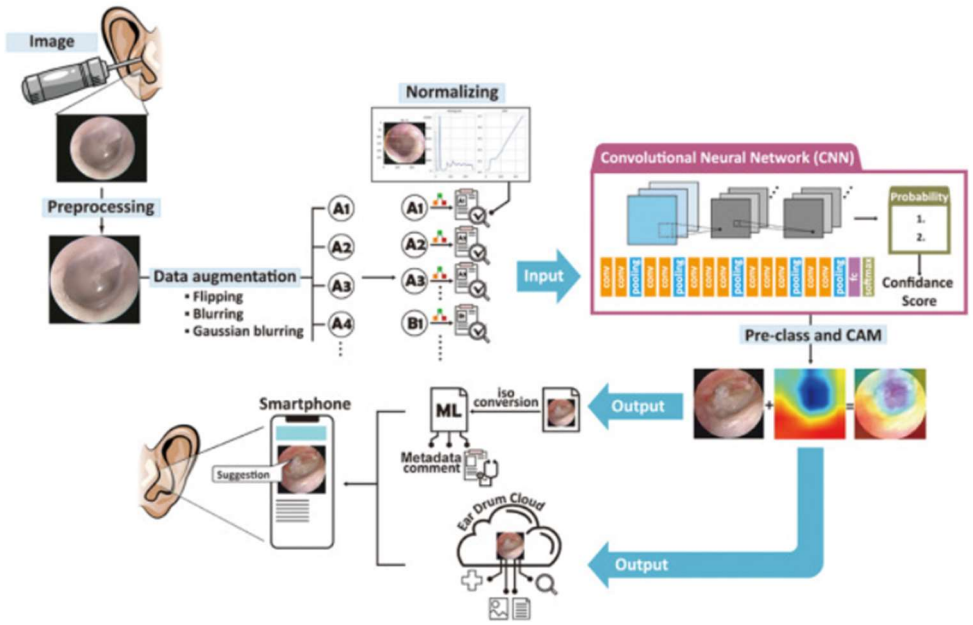


Figure3: Application's Flow Overview

4. DataSet

.

One of the major challenges in our development process has been acquiring a comprehensive and diverse dataset of ear images. Initially, we aimed to classify images into specific conditions such as Otitis Media with Effusion, Acute Otitis Media, and Chronic Otitis Media.

Figure 4: Dataset's Original Division

However, due to the limited availability of high-quality, labeled medical images, we had to adjust our dataset structure.

Our dataset now consists of two broad categories: **Infected** and **Normal**. The dataset contains:

- 415 Infected images
- 590 Normal images

Infected images

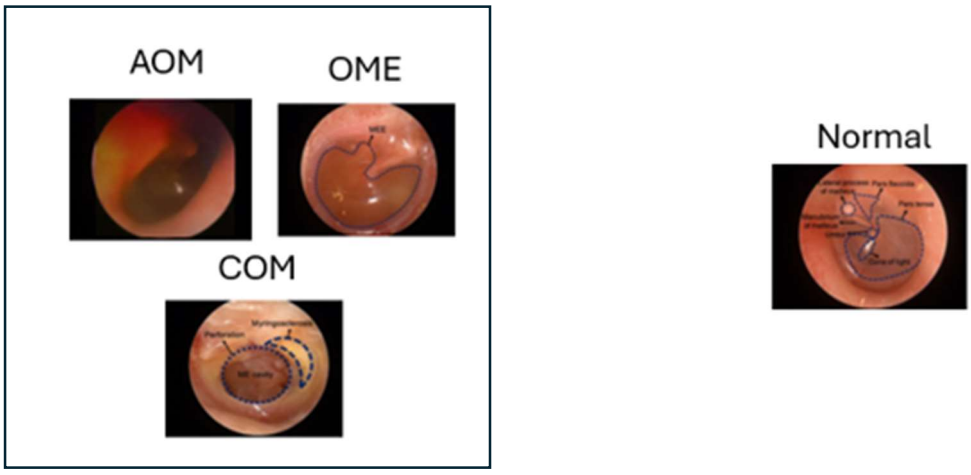


Figure 5: Current Division of Dataset

This limited dataset poses significant challenges in training a robust and accurate machine learning model. To compensate for the class imbalance and improve learning efficiency, we applied **class weighting** during training.

However, the overall size of the dataset remains small, which restricts model generalization and may impact performance in real-world applications. Moving

forward, acquiring a larger and more diverse dataset will be critical. A potential solution is to establish collaborations with healthcare institutions, clinics, or research organizations to collect more annotated images. Expanding the dataset will be essential for enhancing model accuracy and ensuring reliable diagnosis across different populations.

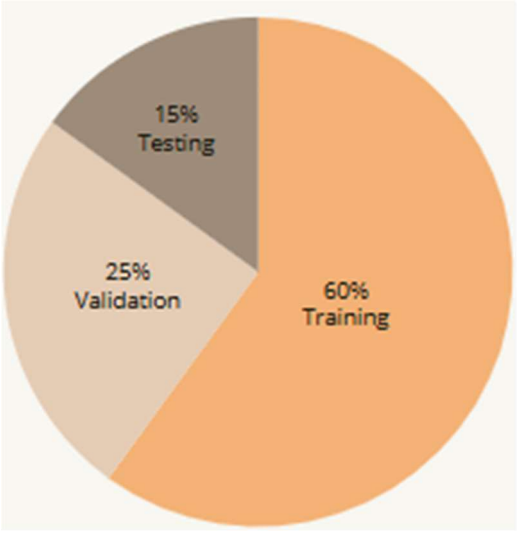


Figure 6: Dataset division into Train, Validation and Test

5. Technologies

.

The development of the *AutoScope* application relies on several cutting-edge technologies and frameworks, each playing a critical role in ensuring the app's functionality, usability, and efficiency. Below is an overview of the primary technologies utilized in the project.

5.1 Kivy

Kivy is a Python-based open-source framework for developing multi-platform applications with natural user interfaces. It is the backbone of the *AutoScope* application's user interface, enabling the development of an intuitive and responsive GUI. With its support for touch-based interactions and cross-platform compatibility, Kivy allows the app to function seamlessly on all devices while maintaining a clean and user-friendly design.

5.2 Flask

Flask is a lightweight web framework used for the backend development of the *AutoScope* app. It handles communication between the mobile app and server, processes image data, and interfaces with the trained TensorFlow model. Flask's simplicity and flexibility make it an ideal choice for managing the app's backend services, including routing, data handling, and API integration.

5.3 OpenCV

OpenCV is an open-source computer vision library that facilitates real-time video capture and processing. In the *AutoScope* app, OpenCV is employed for handling video streams from USB-connected otoscope cameras. It processes the video frames, allowing users to view live feeds and capture images of the ear canal. Its efficient processing capabilities make it a crucial component of the app's functionality.

5.4 TensorFlow

TensorFlow is a powerful open-source machine learning framework. In the *AutoScope* app, TensorFlow is used to integrate a deep learning model for analyzing captured ear images and detecting signs of infection. The model, trained on labeled datasets, predicts the likelihood of infections and provides diagnostic insights. TensorFlow's flexibility and scalability make it ideal for this complex image classification task.

6. Environments

.

The development and deployment of the *AutoScope* application required the use of several environments tailored to specific aspects of the project. These environments were chosen for their ease of use, compatibility with the project's technologies, and efficiency in handling development and testing tasks. Below is an overview of the key environments.

6.1 Visual Studio Code

Visual Studio Code (VS Code) served as the primary integrated development environment (IDE) for both server-side and client-side development of the *AutoScope* application.

- **Server Development**: Flask-based backend services were developed in VS Code. The IDE's extensions, such as Python, provided essential tools for writing and debugging backend code, including API development and integration with Firebase.
- **Client Development**: The Kivy-based GUI and its integration with the USB otoscope and TensorFlow model were implemented and tested in VS Code. Its debugging features, syntax highlighting, and Git integration streamlined the development process.

VS Code's versatility and lightweight design made it an ideal choice for managing the various components of the project within a unified workspace.

6.2 Google Colab

Google Colab was essential for handling tasks that required high computational resources, particularly the training of the TensorFlow-based deep learning model. Its GPU capabilities enabled efficient processing of large datasets, which would have been impractical on local machines without expensive dedicated hardware. By utilizing Colab, we significantly reduced the time and cost associated with model training while ensuring that the computational requirements were met. Additionally, Colab's collaborative features allowed team members to work together seamlessly by sharing and reviewing code in real time, enhancing productivity and ensuring that all contributions were integrated effectively into the project. This platform was chosen not only for its performance advantages but also for its ability to support a distributed and collaborative development process.

6.3 FireBase

Firebase provided a cloud-based environment for securely managing user data and supporting essential application features.

- **Authentication**: Firebase Authentication was used to handle user sign-ins and account management securely.
- **Cloud Storage**: Captured images and analysis results were stored in Firebase Storage, ensuring data accessibility and durability.
- **Real-Time Database**: Firebase Realtime Database facilitated efficient storage and retrieval of metadata, such as user activity logs and settings.

Firebase's integration with the application backend, developed in Flask, enabled seamless data flow between the client and server, ensuring a consistent user experience.

6.4 PythonAnyWhere

PythonAnywhere served as the hosting environment for the server-side components of the *AutoScope* application. Its ease of deployment and support for Python-based web applications made it an excellent choice for managing the backend.

- **Flask Hosting**: The Flask-based server was deployed on PythonAnywhere, enabling secure API endpoints for communication between the client application and the backend services.
- **Always-On Availability**: With PythonAnywhere's always-on functionality, the backend services remained available 24/7, ensuring that users could access the application at any time.

PythonAnywhere’s web console, built-in scheduler, and error logging tools were essential for debugging, maintaining, and scaling the backend infrastructure. Its simplicity and robust feature set allowed the team to focus on application development rather than server management.

7. System Architecture

.

This section provides a comprehensive overview of the *AutoScope* application architecture, using UML diagrams to illustrate the structure and workflow. A package diagram highlights the modular design, emphasizing the separation of the client, backend, and machine learning components. Activity diagrams with swim-lane partitioning depict the sequence of interactions between the user, mobile application, and backend, showcasing responsibilities for each system component.

7.1 System Package Diagram

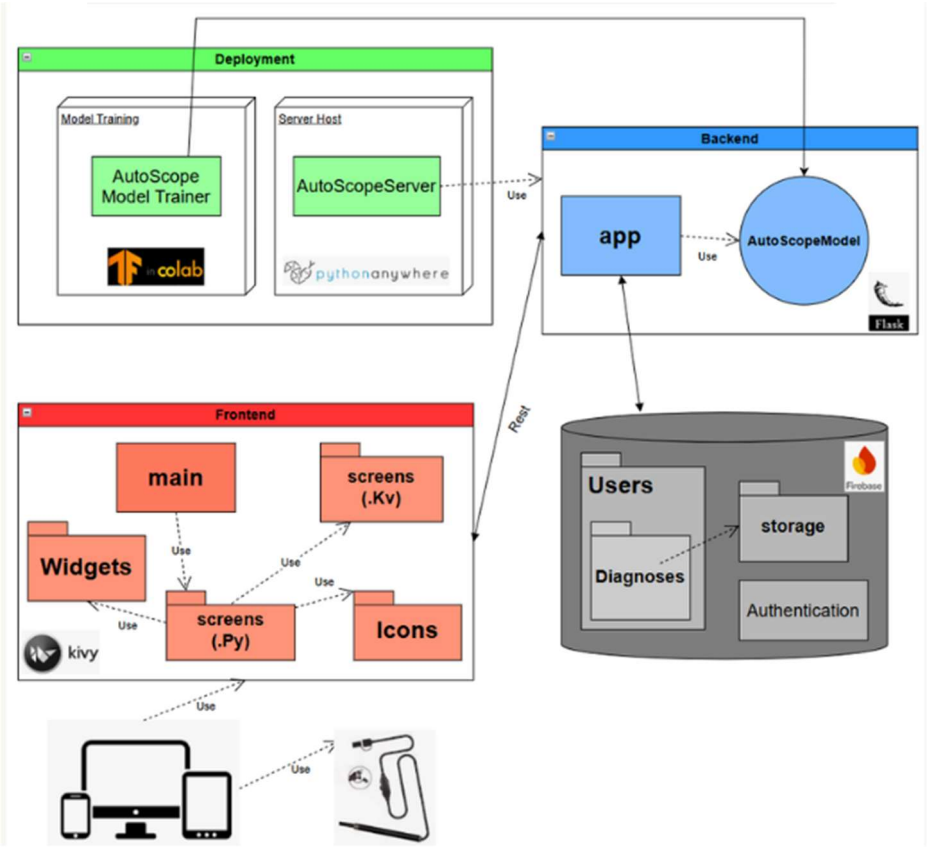


Figure7: System Architecture with detailed package diagram

7.1.1 Frontend

The frontend layer is developed using *Kivy* and serves as the mobile application interface. It manages user interactions, visual presentation, and integration with external hardware like the USB-connected otoscope. The frontend is organized into the following components:

- **Main Object**: Acts as the central controller for the frontend flow, managing transitions between screens and initializing core functionalities.
- **Screens Packages**:
  Kivy Files: Responsible for defining the visual styling and layout of screens.
  Python Files: Contain the logic and interactions for each screen, such as user input handling and data communication with the backend.
- **Icons Package**: Stores app icons and graphical elements used throughout the interface.
- **Widgets Package:** Contains commonly used UI widgets for Kivy framework

7.1.2 Backend

The backend is implemented in Flask and provides the server-side functionality of the application. All core functionalities, including user authentication, history retrieval, and AI model inference, are centralized within app.py, ensuring a streamlined and modular structure.
The app.py file serves as the centralized controller, handling incoming requests and coordinating backend processes. It includes:

- **User Authentication**: Manages user registration, login, and authentication, ensuring secure access to the application.
- **User Results**: Retrieves and displays previously analyzed images stored in the database, allowing users to review past results.
  Save new results and diagnosis to database.
- **App Sharing via Email**: Users can share the application with others by sending an invitation email directly from the app. This feature streamlines accessibility and encourages broader adoption.
- **AI Model Integration**: The trained ResNet50 model is directly integrated within app.py, processing uploaded images and returning predictions about potential ear infections.

By consolidating all functionalities within app.py, the backend maintains a simple yet effective structure, reducing complexity while ensuring scalability and maintainability.

7.1.3 Database

The *Firebase* database serves as the central storage and management solution for *AutoScope*. Its structure includes:

- **Authentication Functionality**: Ensures secure login and user verification.
- **Users Collection**: Stores user profiles and their associated data, such as analysis results and personal details. It includes a collections of user's history.
- **Storage**: A dedicated area for saving image history, including both raw images and processed analysis results, organized by user profiles.

7.1.4 Deployment

The backend server is deployed on PythonAnywhere, chosen for its reliability and ease of use in hosting Python-based applications. This server plays a critical role in the functionality of *AutoScope* by managing communication between the desktop application and the image analysis model. When users upload or capture images through the desktop interface, these images are sent to the backend server, where the deep learning model processes them to detect potential signs of infection. The server then returns the results to the desktop application for display to the user. Deploying the backend on a dedicated platform like PythonAnywhere ensures that the application can handle server requests efficiently, maintain a consistent connection with the desktop app, and provide a seamless experience for users. Additionally, PythonAnywhere's scalable infrastructure supports future growth, allowing for enhancements to the model, expansion of features, and handling of larger datasets without compromising performance.

7.2 Activity Diagrams

This chapter highlights the primary activities involved in the *AutoScope* application, illustrating the step-by-step workflows and interactions between the user and the system. For each activity, we use UML activity diagrams to provide a structured visual representation of the processes. These diagrams employ swim-lane partitions to clarify the responsibilities of the frontend, backend, and database components, ensuring a clear understanding of the system's behavior.

7.2.1 Analyzing captured image from Otoscope

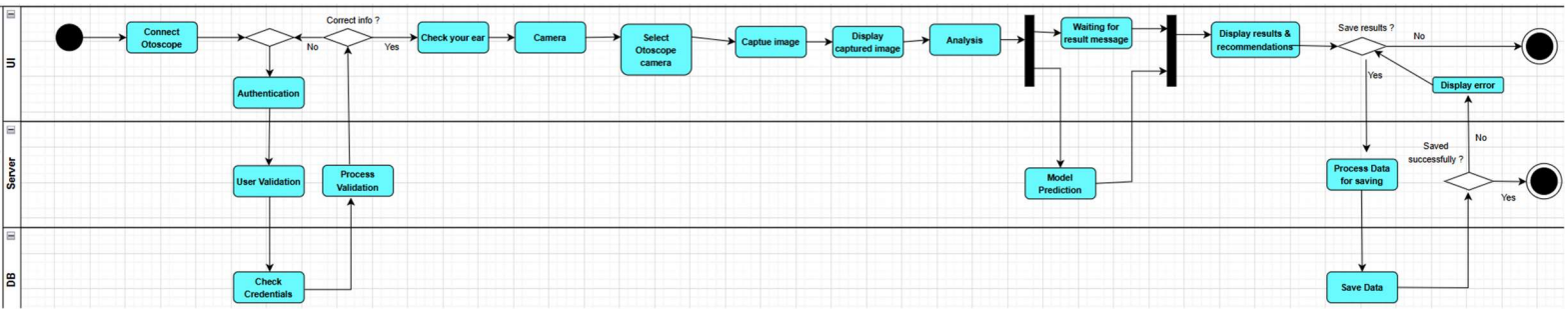


Figure 8:Activity diagram for Analyze eardrum image taken from digital Otoscope, using our application

This activity diagram illustrates the process of taking an image using the AutoScope and analyzing it. The flow begins with user authentication, validated by the server and database. Once authenticated, the user connects the otoscope, captures an image, and views it on the UI. The image is then analyzed by the backend, and the results are displayed. The user can choose to save the results, which are processed and stored in the database. The diagram highlights key decision points, error handling, and the interaction between the UI, server, and database.

7.2.2 Analyzing image from storage

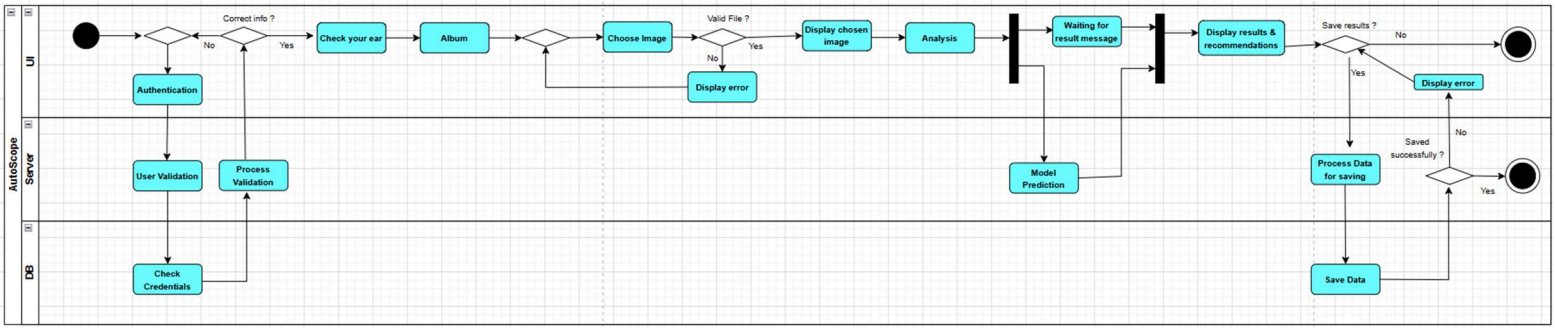


Figure 9 : Activity diagram for Analyzing eardrum image taken from user's storage

This activity diagram represents the process of selecting an image from phone storage and analyzing it. The flow starts with user authentication, validated by the server and database. After successful login, the user navigates to their album to choose an image. If the selected file is valid, it is displayed on the UI and sent for analysis by the backend. The analysis results are then displayed to the user, who can choose to save the results. Saved data is processed and stored in the database.

7.2.3 View Diagnosis history

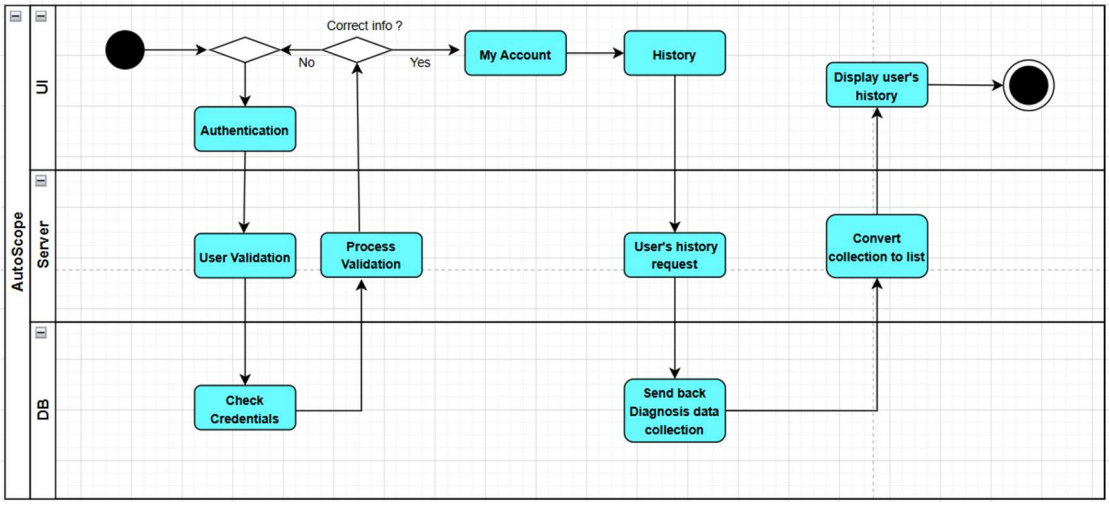


Figure10: Activity diagram for viewing user's history

This activity diagram illustrates the process of viewing the user's history in the *AutoScope* application. The flow begins with user authentication, validated by the server and database. Upon successful login, the user accesses their account and navigates to the history section. A history request is sent to the server, which retrieves

the diagnosis data collection from the database. The server processes the data, converting it into a list format, which is then displayed to the user on the UI, completing the flow.

7.3 Application Interface

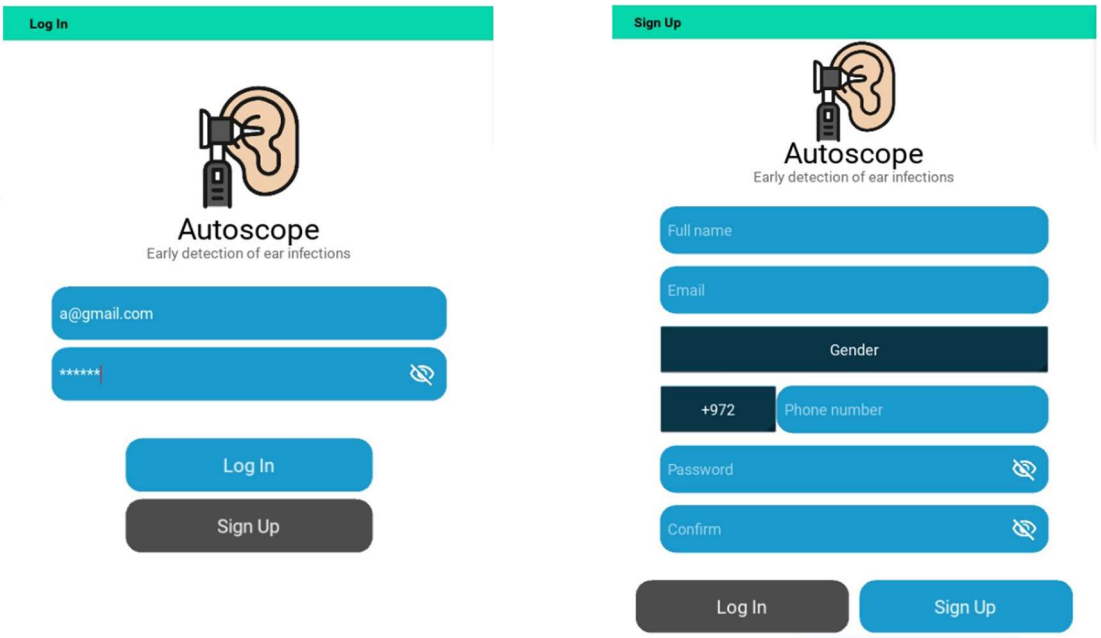


Figure11: Application's Log-In and Sign-Up screens

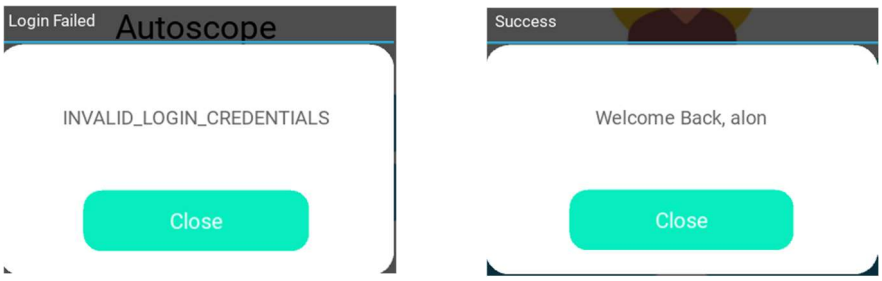


Figure12: Successful and Failed Login messages

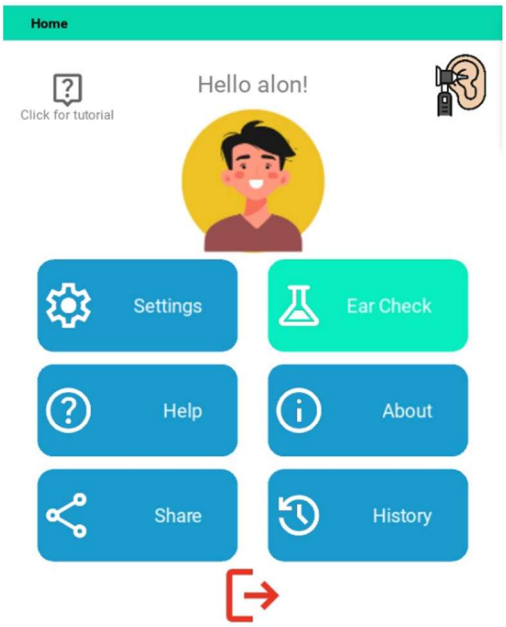


Figure13: Home screen contains all application's functionalities and options.

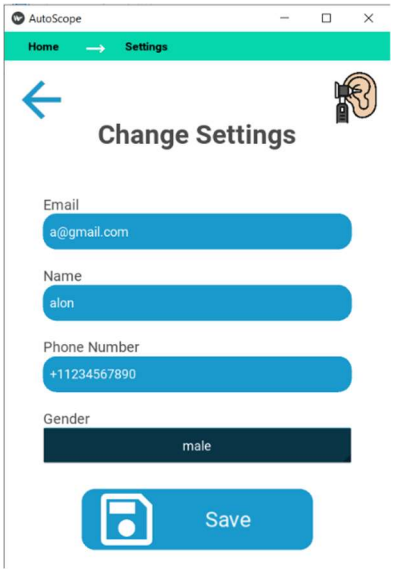


Figure14: Settings screen, where user can update his settings and save changes

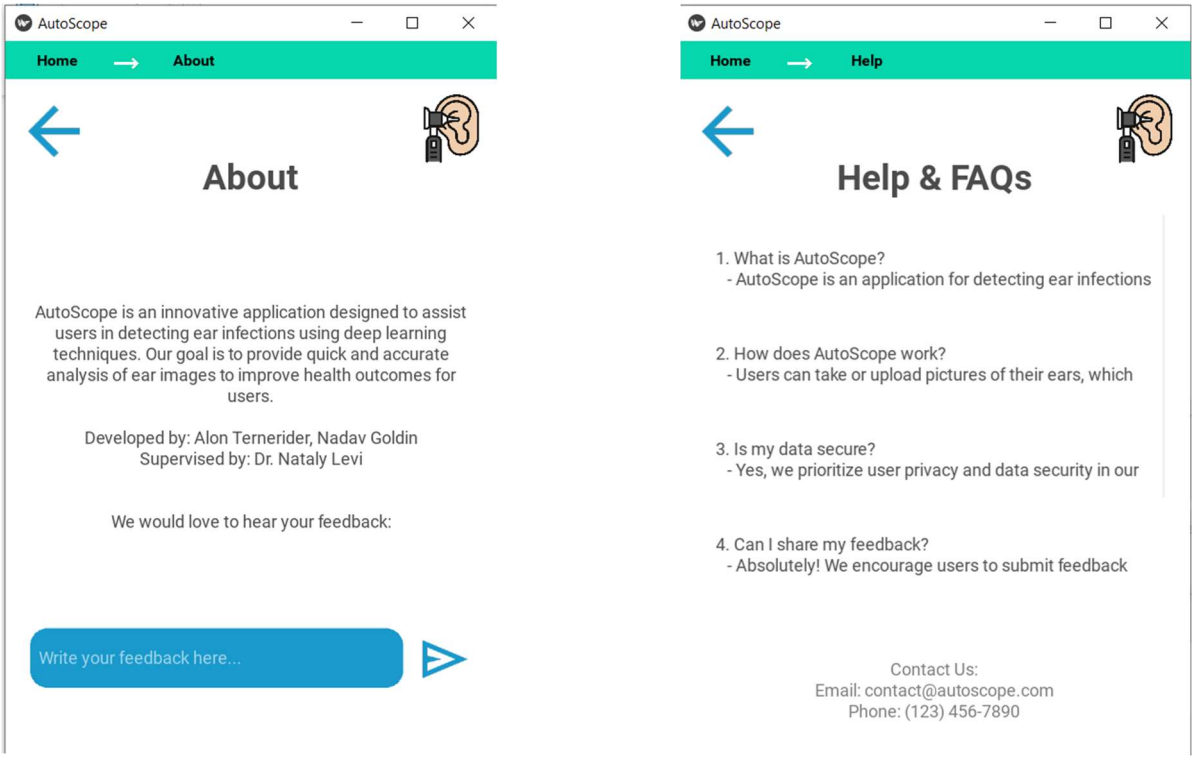


Figure 15: About and Help screens.
user can find important information about the system and how to use it, and leave feedback

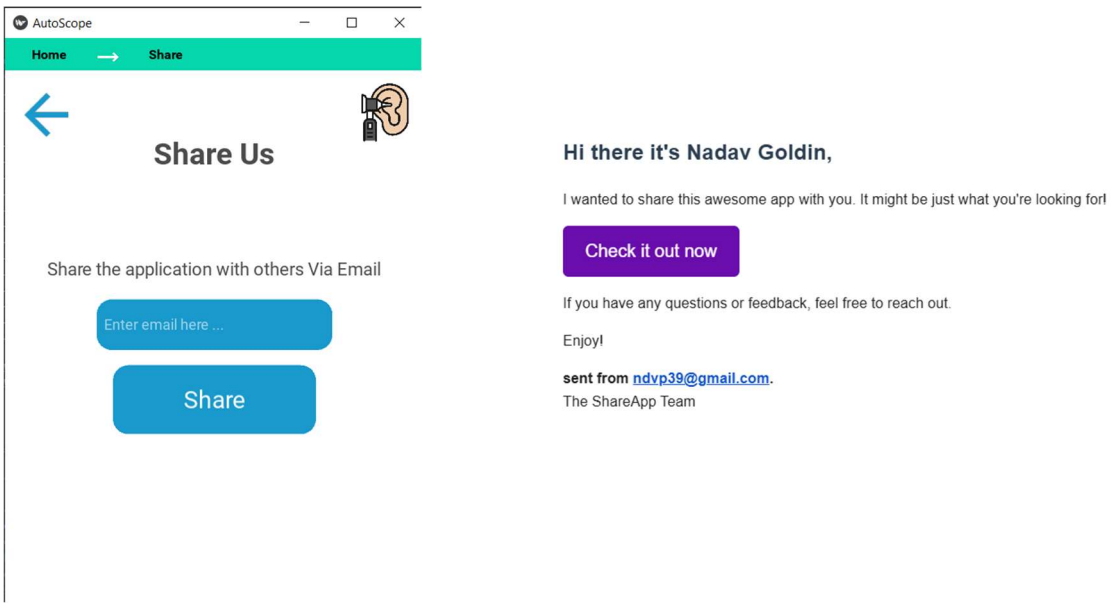


Figure16: User can share the application via email in Share screen. The application will automatically send email message with our GitHub page

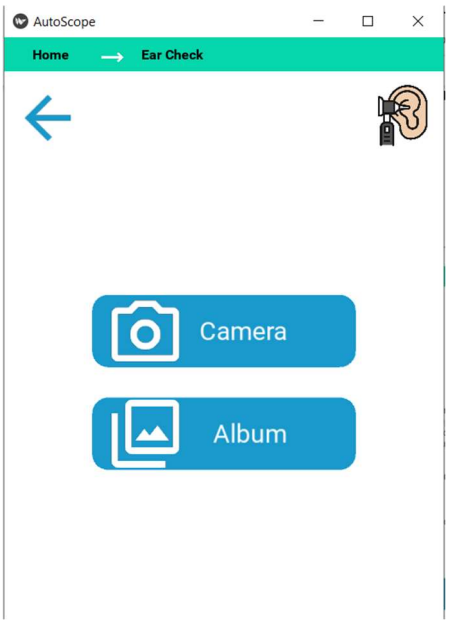
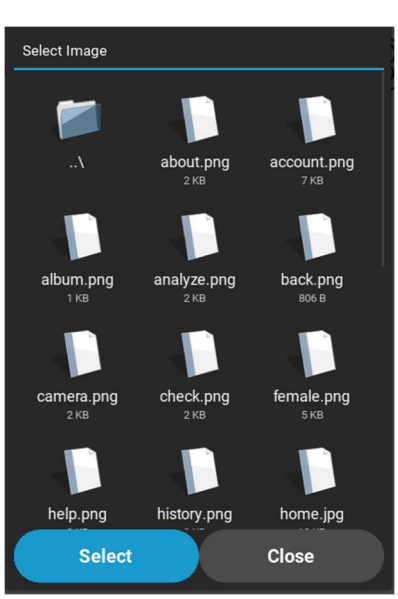


Figure17: To analyze eardrum image, user will have to choose a method to get the required image to process



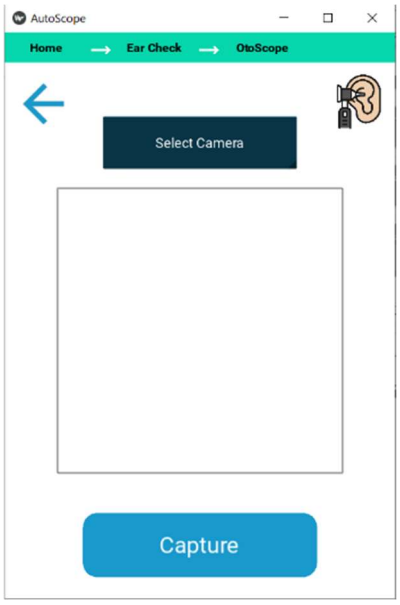


Figure18: User can either choose an image from storage or use a connected camera (otoscope) to capture an image.

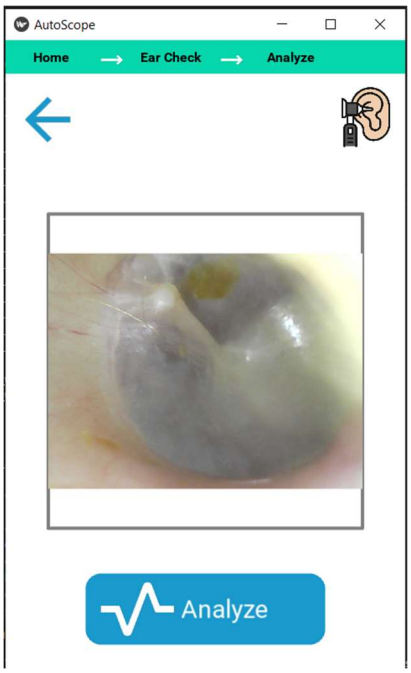


Figure19: After choosing image to analyze, Analyze screen will appear and to process the image, user will have to press Analyze button

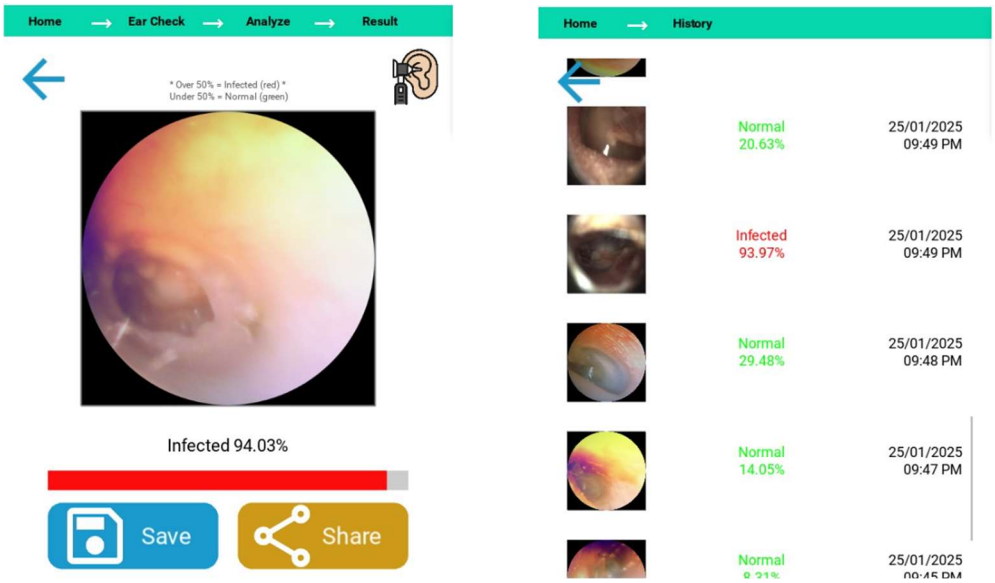


Figure 20: Model's output on the chosen image will appear on the screen. User can save the results and later view it on History screen.

8. Challenges and Obstacles

.

Developing a mobile application like *AutoScope* presented a variety of challenges spanning data acquisition, technical limitations, and deployment complexities. From sourcing high-quality training data to navigating platform-specific restrictions and managing the iterative development process, each stage required creative problem-solving and adaptability. These challenges highlight the technical and logistical hurdles involved in building a robust, user-friendly solution for medical image analysis.

8.1 Data Availability

Data availability is a significant issue, particularly when it comes to acquiring images of eardrums in the early stages of infection. Publicly available datasets often lack diversity or the specific characteristics necessary for training an accurate deep learning model. Such data is rare and hard to find due to privacy concerns and the specialized nature of the medical field.
To address this, synthetic images were programmatically generated by simulating features commonly observed in early infections, such as swelling and redness. This approach helped expand the dataset and improve model robustness, but it also introduced the challenge of ensuring that the synthetic data realistically represents true medical conditions.

8.2 Deployment

Deploying a server capable of running a fully trained model also presents significant challenges. Deep learning models are computationally intensive and require robust infrastructure to operate efficiently. Deploying such models for real-time use can strain server resources and lead to performance issues.
To address this, TensorFlow Lite was integrated into the project. TensorFlow Lite offers a lightweight, optimized version of the model that can run on mobile devices with reduced computational overhead. While this approach simplifies deployment and improves speed, it also requires adapting the model to fit the constraints of TensorFlow Lite, which may involve trade-offs in terms of accuracy and functionality.

8.3 Development

The development process introduces a unique layer of complexity. Integrating the USB otoscope with the application requires careful handling of hardware communication protocols, which can vary between devices and operating systems. Debugging hardware-related issues often necessitates direct testing with the otoscope, making it essential to have consistent access to the device during development. Additionally, as new features or screens are added to the application, ensuring seamless interaction between the software and the otoscope demands iterative testing and refinement. This process can be time-consuming, as errors may only manifest

under specific hardware configurations or during real-time usage, requiring thorough troubleshooting and detailed analysis to resolve.

9. Mobile Integration

.

Initially, we aimed to develop *AutoScope* as a mobile application to leverage the convenience and accessibility of smartphones. However, the development process faced a significant hurdle: Buildozer, the tool required to package Kivy applications for mobile platforms, does not fully support OpenCV integration. Since OpenCV is a core component of our image analysis functionality, this limitation made it impractical to proceed with a mobile-first approach. As a result, we shifted our focus to creating a desktop application, allowing us to maintain the required functionality without the constraints imposed by Buildozer.
Before switching to developing desktop application, we struggled with some mobile integration challenges:

9.1 Building runnable APK

Building an APK for the Kivy-based mobile application is a complex and time-consuming process. Unlike development platforms such as Android Studio, which streamline the APK creation process through automated tools and GUI-based workflows, Buildozer requires extensive manual configuration.
Setting up Buildozer involves creating a proper build environment, resolving dependencies, and handling issues that arise during compilation, such as compatibility with Python packages. This process often becomes a bottleneck in the development pipeline, as even minor changes to the application require a complete rebuild. The lack of intuitive tools further complicates the process, making it less developer-friendly compared to more mature ecosystems.
Google Colab was used for executing *Buildozer* scripts to package the Kivy application into an APK file.
The output is a distributable APK that can be installed on Android devices.

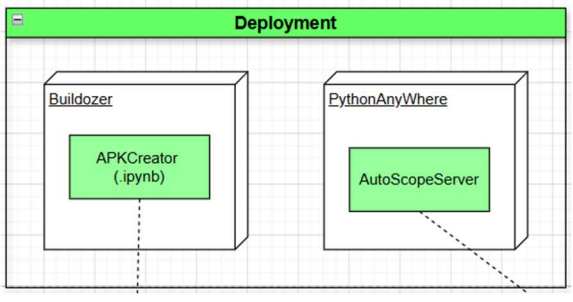


Figure 21: Deployment structure when Building apk using Buildozer

9.2 Hardware Permissions on iOS

Hardware permissions pose another challenge, especially for iOS devices. iOS has strict policies and sandboxing mechanisms that prevent third-party code from accessing hardware directly without explicit permissions. This creates a significant limitation for the application, as the USB-connected otoscope requires direct hardware access to function.
Currently, this restriction means the application is not compatible with iOS devices, limiting its accessibility and reducing its potential market reach. Overcoming this would require extensive collaboration with Apple's ecosystem or finding alternative approaches, such as using wireless communication instead of USB.

10. Testing and Evaluation

.

This section focuses on assessing the performance and reliability of the *AutoScope* application. This includes two key components: an evaluation of the deep learning model's accuracy, precision, and robustness in detecting ear infections, and thorough testing of the application's functionality, usability, and compatibility across various devices. Together, these evaluations ensure that *AutoScope* meets high standards for both technical excellence and user experience.

10.1 Model Evaluation

The evaluation of the deep learning model integrated into *AutoScope* was conducted using a dataset of annotated ear images and the model architecture implemented in TensorFlow. The model's performance was assessed using key metrics, including accuracy, precision, recall, and F1 score, to ensure its reliability in detecting ear infections.
During evaluation, the dataset was split into training, validation, and test sets to prevent overfitting and ensure generalization. The model was trained using several epochs, with real-time monitoring of the loss and accuracy on the validation set. Google Colab environment enabled visualization of performance trends, including a confusion matrix to identify potential misclassifications.

Figure 22: Model's Training session results, showing the training and validation accuracy and loss

The graph on the left shows the training accuracy (blue line) steadily increasing from 0.55 to 0.90 over 10 epochs, while the validation accuracy (orange line) shows a less stable increase from 0.55 to 0.85. The graph on the right shows the training loss (blue line) steadily decreasing from 0.8 to 0.2, while the validation loss (orange line) shows a less stable decrease from 0.8 to 0.35.

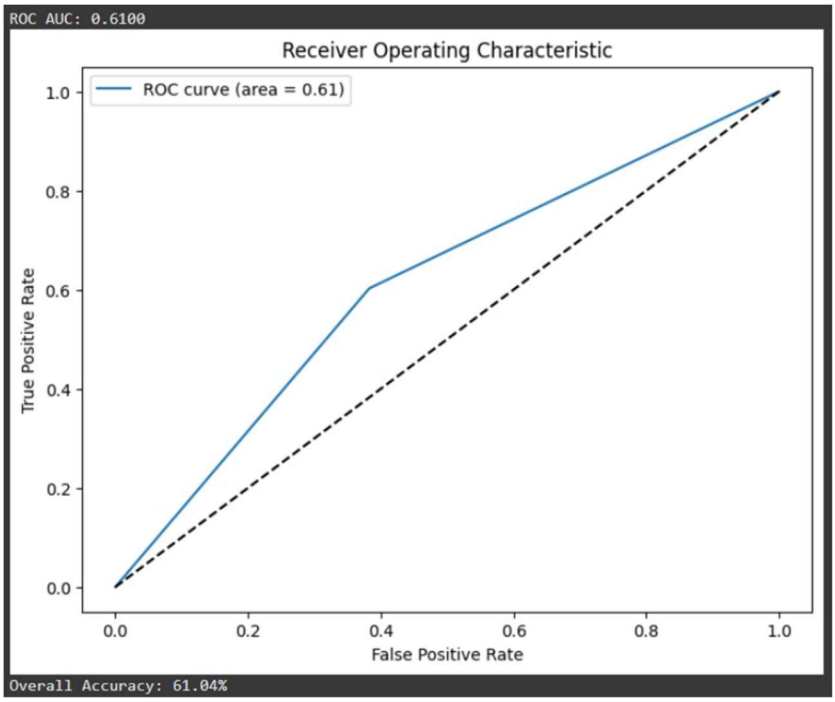


Figure23: Model's ROC curve, indicating the overall accuracy of the trained model

This image depicts a Receiver Operating Characteristic (ROC) curve, which illustrates the diagnostic ability of the binary classifier system. The ROC curve with an area under the curve (AUC) of 0.61 indicates the classifier's performance. The overall accuracy of the classifier is noted as 61.04%.

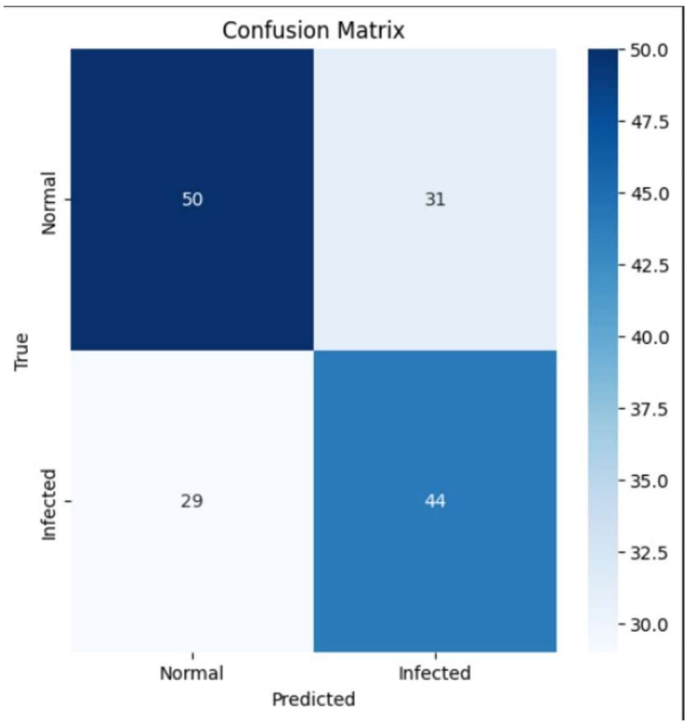


Figure 23: Confusion matrix, helping us analyze the trained model's accuracy on different cases

This confusion matrix highlights both the strengths and weaknesses of the model. It shows the model's ability to correctly identify infected and normal cases, while also indicating areas for improvement.

while the model demonstrated high precision and strong recall, the limited dataset posed challenges in achieving optimal performance. These results highlight the importance of using a more extensive dataset for training to improve the model's generalization and overall accuracy. The model's readiness for integration into the *AutoScope* application is promising, but further refinement with a larger dataset is necessary to ensure reliable and efficient image analysis.

10.2 System Testing

System testing is a critical phase in ensuring that the application operates as intended across various functionalities, providing a smooth and reliable user experience. This involves testing the system’s performance, usability, and compatibility under different scenarios to identify and fix issues before deployment. The testing process not only validates the functionality but also ensures that the system meets user expectations. A combination of automated, manual, and user-centric testing methods were employed throughout the development lifecycle to refine the application further.

1. **User Feedback During Development**
   One of the most effective methods involved sharing the application with potential users during the development process. This iterative approach enabled users to interact with the application in real-world scenarios and provide valuable feedback on its usability, accuracy, and interface. Insights from this feedback helped address potential issues early on, improving both functionality and user satisfaction.

2. **Stress Testing for Performance**
   The application was subjected to stress testing to evaluate its performance under various conditions. This included uploading multiple images simultaneously, connecting and disconnecting the otoscope repeatedly, and simulating prolonged usage. The aim was to ensure the app remained responsive and functional even under heavy load, confirming its stability for real-world use.

3. **Edge Case Testing**
   The application was tested with a variety of edge cases to evaluate its robustness. Examples include uploading corrupted or unsupported image files, simulating device disconnection during image analysis, and testing with low system resources. These tests ensured the app could gracefully handle unexpected scenarios without compromising performance or user experience.

11. References

1. TytoCare website. https://www.tytocare.com

2. CellScope. https://www.engineeringforchange.org/solutions/product/oto

3. HearScope. https://www.hearxgroup.com/hearscope

4. Ken Kitamura, Yukiko Iino, Yosuke Kamide, Fumiyo Kudo, Takeo Nakayama , Kenji Suzuki,
   Hidenobu Taiji, Haruo Takahashi, Noboru Yamanaka, Yoshifumi Uno. "Clinical Practice Guidelines for the diagnosis and management of acute otitis media (AOM) in children in Japan", September 18, 2014. https://www.sciencedirect.com/science/article/pii/S0385814614001692?casa_token=OMc6KuvylDkAAAAA:g2oGfSeDWeXb1yi9ZwBl4s7yrZEmj0dYSR-VJxXkaaawiNqsAdieT8xxbqfN4Htf08GZm3p8MQ.

5. KATHRYN M. HARMES, MD; R. ALEXANDER BLACKWOOD, MD, PhD; HEATHER L. BURROWS, MD, PhD; JAMES M. COOKE, MD; R. VAN HARRISON, PhD; and PETER P. PASSAMANI, MD. "Otitis Media: Diagnosis and Treatment", October 1, 2013. https://www.aafp.org/pubs/afp/issues/2013/1001/p435.pdf.

6. STEVEN L. SOLOMON, MD, AND KRISTEN B. OLIVER. "Antibiotic Resistance Threats in the United States: Stepping Back from the Brink", June 15, 2014. https://www.aafp.org/pubs/afp/issues/2014/0615/p938.html.

7. Benjamin D. Brooks, Amanda E. Brooks. "Therapeutic strategies to combat antibiotic resistance", October 28, 2014.

https://www.sciencedirect.com/science/article/pii/S0169409X1400235X?casa_token=i-k98uHdfUIAAAAA:f-s1okg0-j3PRT0ldHaFDBVU9OBwELfD1SD63Ym08TvQPtAKqXW5JDvt89fcP1Wmf5SKc0Krxw.

8. Mayo Clinic Stuff. "Ear infection (middle ear)", June 23, 2021. https://www.mayoclinic.org/diseases-conditions/ear-infections/symptoms-causes/syc-20351616.

9. Jerome O Klein. "The burden of otitis media", January 10, 2001. https://www.sciencedirect.com/science/article/pii/S0264410X00002711.

10. Daniele Ravì, Charence Wong, Fani Deligianni, Melissa Berthelot, Javier Andreu-Perez, Benny Lo, Guang-Zhong Yang. "Deep Learning for Health Informatics", December 29, 2016. https://ieeexplore.ieee.org/abstract/document/7801947.

11. Dinggang Shen, Guorong Wu, Heung-Il Suk. "Deep Learning in Medical Image Analysis", March 9, 2017. https://www.annualreviews.org/content/journals/10.1146/annurev-bioeng-071516-044442.

12. Stephen M .Pizer, E. Philip Amburn, John D. Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart ter Haar Romeny, John B. Zimmerman, Karel Zuiderveld. "Adaptive histogram equalization and its variations", September 29, 2007. https://www.sciencedirect.com/science/article/abs/pii/S0734189X8780186X?via%3Dihub.

13. Sewoong Kim, Jihun Kim, Minjoo Hwang, Manjae Kim, Seong Jin Jo, Minkyu Je, Jae Eun Jang, Dong Hun Lee, and Jae Youn Hwang. "Smartphone-based multispectral imaging and machine-learning based analysis for discrimination between seborrheic dermatitis and psoriasis on the scalp", January 24, 2019. https://opg.optica.org/boe/fulltext.cfm?uri=boe-10-2-879&id=404293.

14. Thiago C. Cavalcanti ,Sewoong Kim ,Kyungsu Lee ,Sang-Yeon Lee ,Moo Kyun Park ,Jae Youn Hwang. "Smartphone-based spectral imaging otoscope: System development and preliminary study for evaluation of its potential as a mobile diagnostic tool", March 5, 2020. https://onlinelibrary.wiley.com/doi/10.1002/jbio.201960213.

15. Thiago C. Cavalcanti, Hah Min Lew, Kyungsu Lee, Sang-Yeon Lee, Moo Kyun Park, and Jae Youn Hwang. "Intelligent smartphone-based multimode imaging otoscope for the mobile diagnosis of otitis media", November 23, 2021. https://opg.optica.org/boe/fulltext.cfm?uri=boe-12-12-7765&id=465384.

P.K Goel, S.O Prasher, R.M Patel, J.A Landry, R.B Bonnell, A.A Viau, "Classification of hyperspectral data by decision trees and artificial neural networks to identify weed stress and nitrogen status of corn", January 18, 2003. https://linkinghub.elsevier.com/retrieve/pii/S0168169903000206.

Capstone Project Phase B

AutoScope – Application for ear infection detection

24-2-D-24

Authors:
Alon Ternerider
Nadav Goldin

Supervisor: Dr. Nataly Levi

USER GUIDE

GitHub: https://github.com/ndvp39/autoscope

Table Of Contents

Introduction

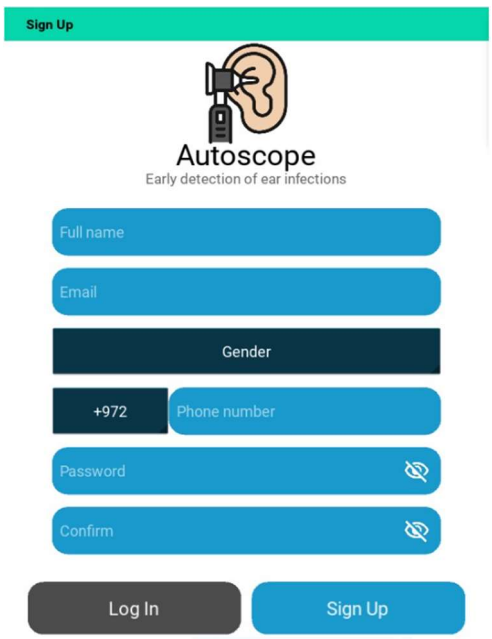
This guide provides a comprehensive explanation of the *AutoScope* app, detailing all the information users need to navigate and use the app effectively.
The guide is divided into several sections, each focusing on the core functionality of the app and how to interact with its various features.
In addition to this written guide, a video tutorial showcasing the app's functionality is available for users at the following link:

AutoScope - User Guide

Requirments

To use the *AutoScope* app effectively, ensure that you have Otoscope Device. The otoscope should be connected to the app via USB for real-time image capturing.
By meeting these hardware requirements, users can optimize their experience and achieve accurate results while using the *AutoScope* app.
It is possible to use the *AutoScope* without otoscope, just make sure you have the required image to process in your storage.

1. Sign-Up Screen

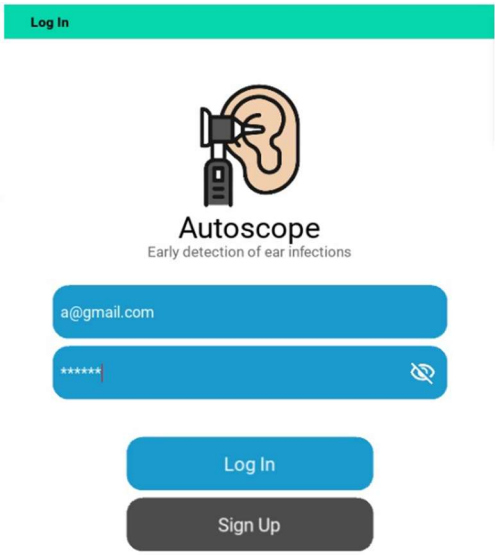


The *Sign-Up Screen* provides a simple and intuitive interface for new users to create an account on the *AutoScope* app. It requires essential user information to ensure a personalized and secure experience. Its key input fields and functionality:

- **Full Name**: Enter your complete name as it will be associated with your profile for personalization and identification purposes.
- **Email**: Provide a valid email address. This is essential for account creation and communication.
- **Gender**: Select your gender from the available options to help tailor app functionality or insights where applicable.
- **Phone Number**: Input your phone number along with the correct country code. This is necessary for account security and potential app notifications.
- **Password**: Create a strong and secure password. Make sure it adheres to security best practices to protect your account.
- **Confirm Password**: Re-enter the same password to confirm it matches and eliminate errors during setup.

The screen also features two key action buttons:
Log In: Redirects users who already have an account back to the login screen.
Sign Up: Finalizes the registration process after all fields are correctly filled.

2. Log-In Screen



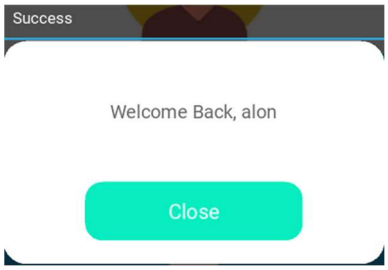
The *Sign-Up Screen* provides a simple and intuitive interface for new users to create an account on the *AutoScope* app. It requires essential user information to ensure a personalized and secure experience. Its key input fields and functionality:

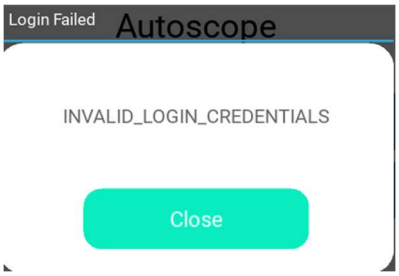
- **Full Name**: Enter your complete name as it will be associated with your profile for personalization and identification purposes.
- **Email**: Provide a valid email address. This is essential for account creation and communication.
- **Gender**: Select your gender from the available options to help tailor app functionality or insights where applicable.

- **Phone Number**: Input your phone number along with the correct country code. This is necessary for account security and potential app notifications.
- **Password**: Create a strong and secure password. Make sure it adheres to security best practices to protect your account.
- **Confirm Password**: Re-enter the same password to confirm it matches and eliminate errors during setup.

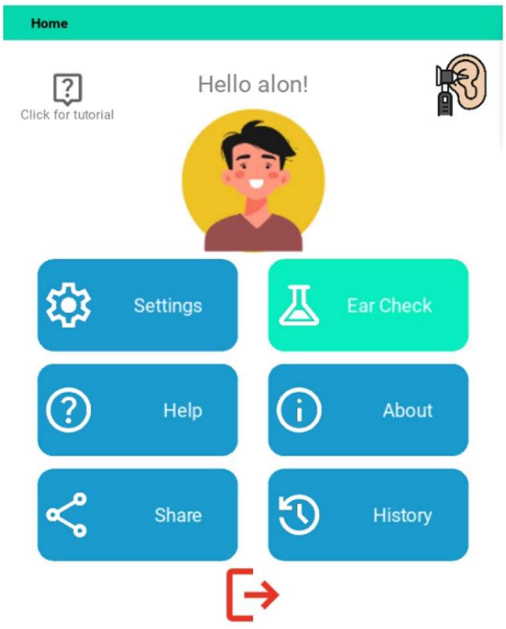
The screen also features two key action buttons:
Log In: Redirects users who already have an account back to the login screen.
Sign Up: Finalizes the registration process after all fields are correctly filled.

3. Login Result Pop-Ups



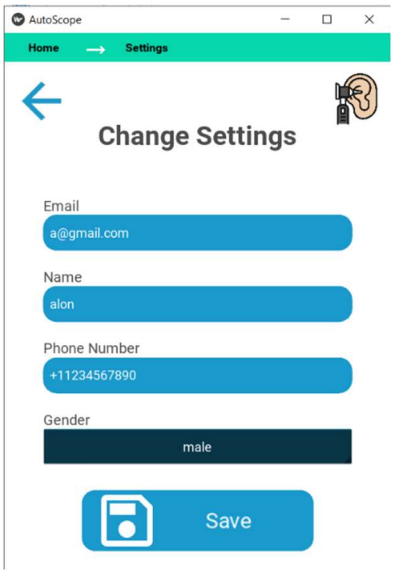


4. Home Page



The *Home Screen* serves as the main navigation hub for the *AutoScope* app, providing users with access to various features and settings. At the top, a personalized greeting, such as "Hello Alon!" and a thematic logo (ear/otoscope), create a welcoming experience.
The screen contains several buttons for navigation: the ***Settings*** button (gear icon) allows users to adjust preferences like notifications, profiles, or otoscope configurations.
The ***Ear Check*** button (flask icon) initiates an ear examination by navigating to the camera or image upload functionality for analysis.
The ***Help*** button (question mark icon) provides access to guides, FAQs, and troubleshooting information, while the ***About*** button (info icon) displays application details, including the version, developers, and privacy policy.
Users can share the app using the ***Share*** button (share icon), which allows sharing through social media, email, or messaging platforms.
The ***History*** button (arrow loop icon) provides access to previously uploaded images and analysis results.
Finally, the ***Logout*** button (red arrow icon) at the bottom allows users to log out of the app and return to the login screen. This intuitive layout ensures easy navigation between the app's key functionalities.
In addition, to get familiar with the app, user can click on the **tutorial video**, which will open a short video demonstrating the applications functionalities.

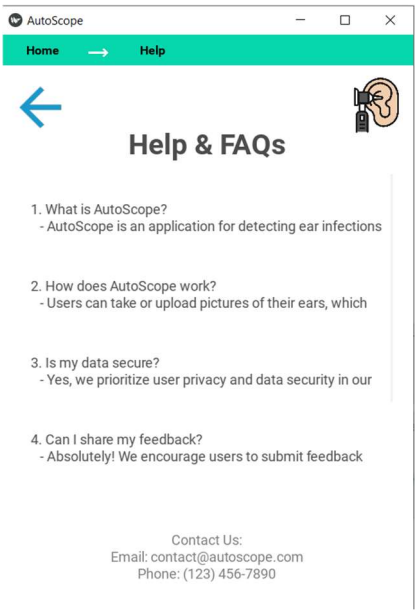
5. Settings Page



The *Settings Screen* allows users to update their personal details and preferences within the app. It displays input fields for the user's **Email**, **Name**, **Phone Number**, and **Gender**, each of which is pre-filled with the current values for easy modification. Users can tap any field to update the information.

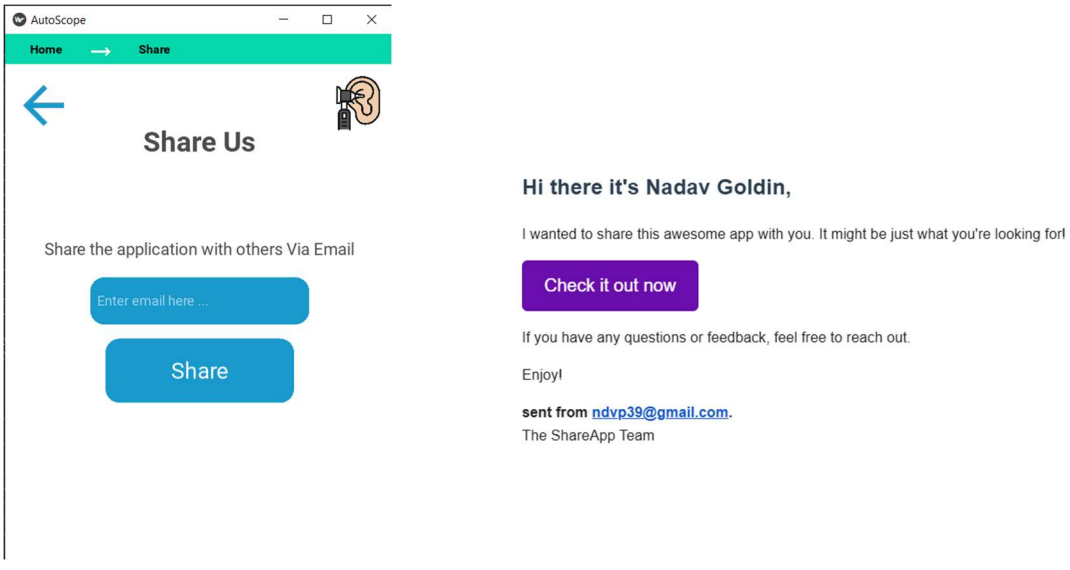
Once changes are made, users press the *Save button* (floppy disk icon) at the bottom to confirm and store the updated details. A *back arrow* in the top left corner provides a way to return to the Home Screen without saving changes.

6. Help Page



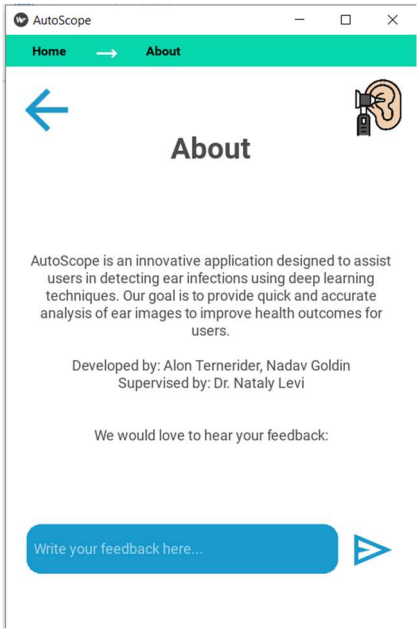
The *Help & FAQs Screen* provides users with answers to common questions about the app and its functionality.
At the bottom, contact information is displayed, including an email address and phone number, for further assistance. A *back arrow* in the top left corner allows users to return to the previous screen.

7. Share Page



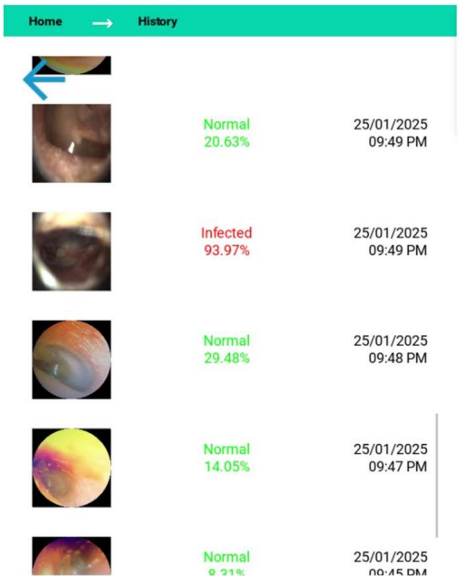
The *Share Screen* allows users to share the app with others via email. It features an input box where users can type the recipient's email address. After entering the email, users can press the *Share button* to send a pre-configured message or link about the app to the specified email. This feature makes it easy for users to recommend *AutoScope* to friends, family, or colleagues.
To the message that is being sent, the project's GitHub page link is attached:

8. About Page



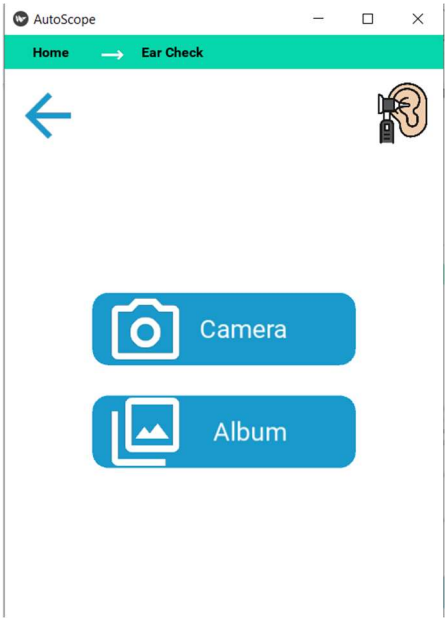
The *About Screen* introduces *AutoScope* as an innovative application designed to assist users in detecting ear infections using deep learning techniques. It highlights the app's goal of providing quick and accurate analysis of ear images to improve health outcomes. The screen credits the development team their supervisor.
Below the introduction, there is a feedback section where users can write their comments or suggestions in a provided input box, followed by a *send button* to submit their feedback.

9. History Page



The *History Screen* provides a clean and organized view of previously analyzed ear images. Each entry displays a thumbnail of the analyzed image, the corresponding result or diagnosis, and a timestamp indicating when the analysis was conducted. *Navigation arrows* at the bottom of the screen allow users to browse through multiple entries, while a *back button* in the top-left corner lets users return to the previous screen.

10. Ear-Check Page

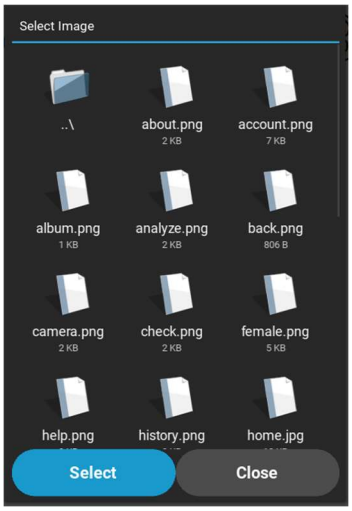


The *Ear-Check Screen* provides two main options for users to proceed with analyzing ear images:

1. **Camera**: Take a picture using an otoscope or a connected camera.
2. **Album**: Select a previously saved image from their device's gallery or album.

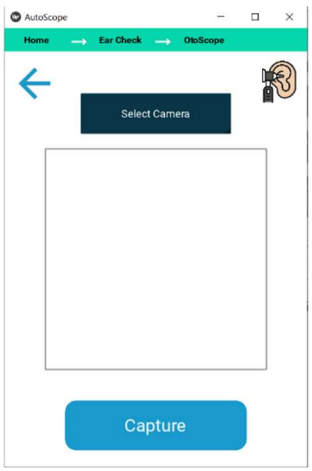
A *back button* on the left allows users to return to the previous screen, and the consistent header bar ensures navigation clarity.

11. File Selection Pop-Up

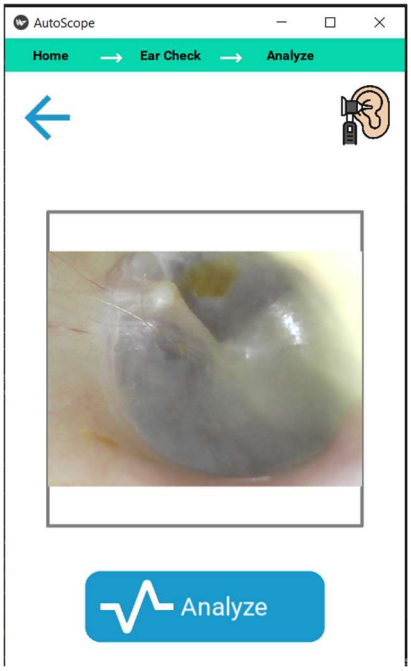


From *ear-check screen*, if user will choose to load image from his storage, a pop-up window will appear with interface similar to File Explorer.
The user will then choose the image he wants to analyze and to move forward with the process, he will press *select button*.
To cancel the file selection process, User will press *Close button*.

12. Otoscope Video Page

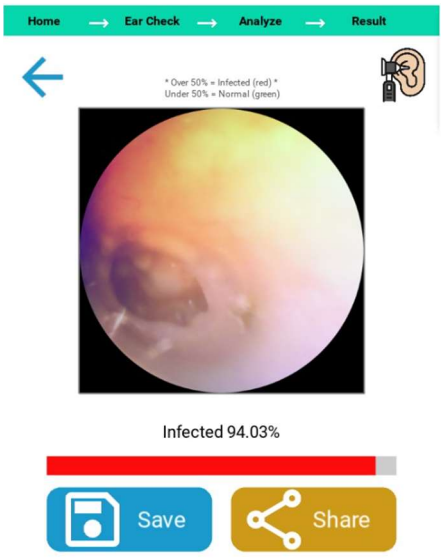


From *ear-check screen*, if user will choose to use camera to take an image, The *OtoScope Video Screen* will apper.
The user will then choose the required camera from the *camera selection box*.
After camera is chosen, a video stream from that camera will begin.
The user is able to Take a snapshot of the current video stream and use that snapshot As the image to analyze:



After freezing the image and to continue with the process, user will click on *Analyze button.* To return to the Ear check method selection, User will choose the *back arrow button*.

13. Result Page



After pressing *Analyze*, our model will process the image and after short delay, the result will appear under the image.
We provide the user a colored bar to indicate the result.
from here, user can save the image and result using *save button* and then he can share the result using *share button*.
When saving the image and result, user can later view them on *History Screen*.

Capstone Project Phase B

AutoScope – Application for ear infection detection

24-2-D-24

Authors:
Alon Ternerider
Nadav Goldin

Supervisor: Dr. Natali Levi

MAINTENANCE GUIDE

GitHub: https://github.com/ndvp39/autoscope

Table Of Contents

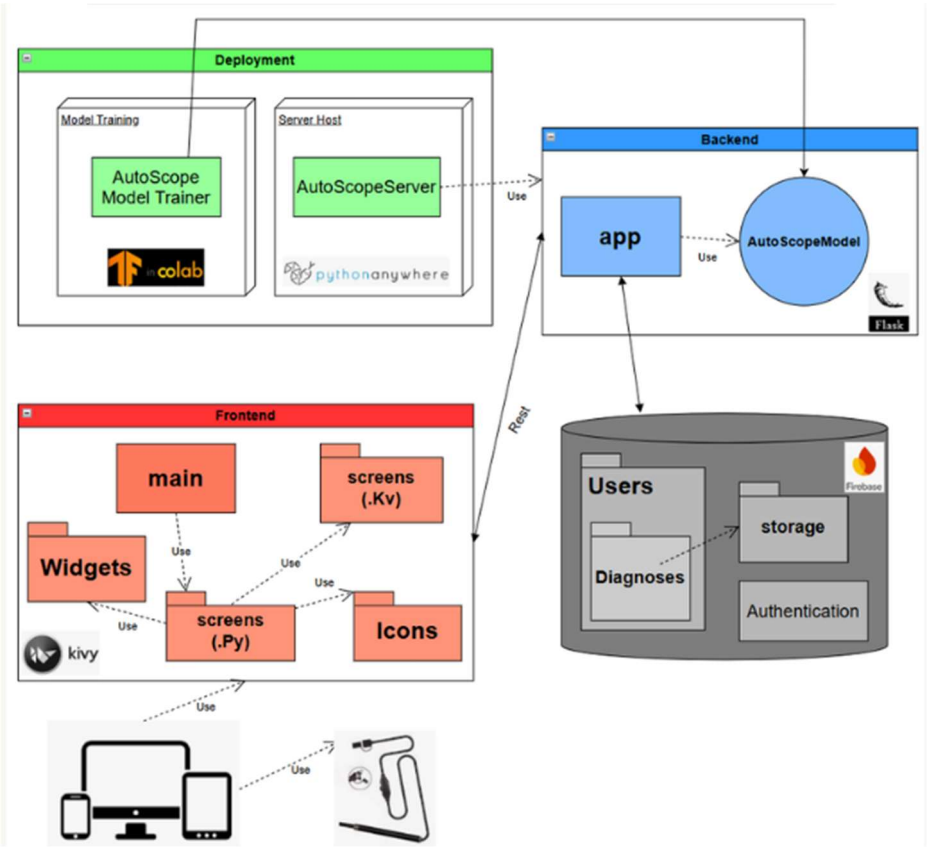
Introduction

This maintenance guide is designed to support engineers tasked with the ongoing upkeep, optimization, and enhancement of the *AutoScope* system. It offers comprehensive instructions for maintaining system reliability, managing dependencies, addressing potential security vulnerabilities, and resolving common technical challenges. Structured into clear and concise sections, this guide ensures that each critical aspect of system maintenance is thoroughly addressed, empowering teams to sustain performance and adapt to evolving requirements effectively.

1. System Architecture Overview

1.1 System Components

Our system employs a client-server architecture and thus has several independent components, as you can see in the following package diagram detailing the system structure:



- **Frontend**
  The frontend is developed using the Kivy framework, a Python-based platform for building cross-platform user interfaces. It handles user interactions, displays results, and communicates with the backend for processing. The interface includes screens for selecting or capturing images, analyzing them, and displaying diagnostic results. Kivy's design ensures the app runs seamlessly on both mobile and desktop devices, providing a user-friendly and responsive experience.

- **Backend**
  The backend is implemented using Flask, a lightweight Python web framework, and is deployed on PythonAnywhere. It manages communication between the frontend, database, and AI models. The backend handles tasks such as user authentication, image preprocessing, and model inference. API endpoints are provided for analysis requests and user data management. The backend also integrates securely with Firebase for storing and retrieving data, ensuring scalability and robust system functionality.

- **DataBase**
  The database uses Firebase, leveraging its built-in authentication functionality for secure user management.
    - User data is stored under the "Users" label, where each user record contains attributes such as gender, phone number, and diagnostic results.
    - Each diagnostic result includes a date, the diagnosis outcome, and a link to the corresponding image stored in Firebase Storage.
    - Images are stored in a dedicated "images" folder, enabling efficient access and retrieval.

  This structure ensures secure and organized data management while supporting scalability for future system growth.

- **Model Training**
  The system integrates *ResNet50* deep learning model. That model is trained using TensorFlow on Google Colab, utilizing cloud-based computational resources. After training and validation, the model is converted to TensorFlow Lite format for deployment within the app, enabling efficient and accurate real-time image classification.

1.2 System Flow

The *AutoScope* system follows a streamlined process that guides users through capturing or uploading an image of the eardrum, receiving diagnostic results, and saving or sharing the outcomes. The typical flow of a successful system run is outlined as follows:

1. **Login\Sing-Up**
   The user begins their experience by either logging into an existing account or signing up to create a new one. The sign-up process ensures that the app has all the necessary user details to personalize their experience. After successful login or registration, the user is directed to the *Home Screen*, which serves as

the central hub of the app.
For new user, it is recommended to watch the tutorial using the link from that screen.
The *Home Screen* includes key options such as Ear Check, Help, About, Share, and History, along with a friendly greeting and user avatar for a personalized touch. If a user is unfamiliar with the app, they can click the *Help* button or the "?" icon, which provides step-by-step instructions on how to use the app's features, ensuring a smooth and intuitive onboarding process.

2. **Initiate Ear-Drum capture**
   To start the diagnostic process, the user clicks the "Ear Check" button on the Home Screen. This is the main function of the app and takes the user to the image selection step. The user is guided seamlessly to choose how they would like to provide the image for analysis, making the process simple and clear.

3. **Choosing Image Source**
   At this step, the user decides how to provide the image for analysis. They are given two options:

   - **Using the otoscope**: This option connects the app to an otoscope device, enabling a live video feed where the user can capture a real-time image of the eardrum.
   - **Selecting an image from storage**: The user can browse their device's storage to select an existing image of the eardrum.
     Only files with specific image extensions are allowed to ensure compatibility. Supported extensions include:
     .png, .jpg, .jpeg, .gif, .bmp, .tiff
     This filtering ensures that users can only select valid image files for analysis, maintaining a smooth and error-free experience.

   This flexibility ensures that users can proceed regardless of whether they have the otoscope on hand or have already taken an image in the past.

4. **Image Capture using Otoscope**
   If the user selects the otoscope option, they are taken to a screen where the otoscope's live video feed is displayed.
   To ensure the otoscope is connected, the user is able to see which camaras devices are connected in real time, using a selection list.
   when the user is able to recognize the otoscope, he can simpky choose it and view the live video feed.
   The user can carefully position the otoscope to view the eardrum and press the "Capture" button to take an image. After capturing, the image is displayed, allowing the user to proceed to the analysis step. This process ensures that the image is of high quality and suitable for accurate diagnosis.

5. **Analyzing image**
   Once the image is captured (via the otoscope) or selected (from storage), the app displays the chosen image along with an *"Analyze"* button. The user can

review the image to ensure it's the correct one and press "*Analyze*" to begin processing the image. The app uses advanced deep learning models (ResNet50, InceptionResNetV2, and VGG16) to analyze the eardrum image and generate a diagnosis. This process is quick and provides the user with detailed results based on the analysis.

6. Saving results

After the analysis is complete, the app presents the diagnostic results to the user. These results are displayed along with additional instructions or suggestions, if applicable. Users have the option to save the results, ensuring that the diagnosis is stored securely in the Firebase database under their account.

The process of saving the result involves the following steps:

- **Image Upload**: The image file is temporarily stored and then uploaded to Firebase Storage under a unique filename.
- **Public URL Generation**: A public URL is created for the uploaded image, making it accessible for reference.
- **Firestore Update**: The app updates the user's record in Firestore with the diagnosis string, the public image URL, and the analysis date-time. Each result is appended to an array of past results stored under the user's profile.

This saved data includes the diagnosis, the date of analysis, and a link to the associated image.
Once the results are saved, the user can share them with others such as a doctor or family member.

7. Additional options

The app includes several supplementary features to enhance user experience and functionality:

- **Viewing History**: The user can access all previously saved results by navigating to the History section. This allows them to track past diagnoses, including associated dates and images, making it easier to monitor their health over time.
- **Settings**: The Settings section allows users to modify their profile details, such as name, gender, phone number, and preferences. This ensures the app remains tailored to their needs.
- **Progress Bar**: A progress bar is displayed at the top of the screen throughout the diagnostic process. This visual guide helps users track their current step and provides clarity on how far they are in completing the sequence.
- **Sharing the App**: Through the "*Share*" button on the Home Screen, users can easily recommend the app to others by sharing a download link via email, helping expand its reach and accessibility.

2. Software Dependencies.

The *AutoScope* app relies on a specific set of software dependencies to function effectively. This section provides an overview of the key libraries and frameworks used in both the backend and the image analysis components of the system. Proper management and regular updates of these dependencies are essential to maintaining system stability and performance.

- **Python**
  The app is built on Python version 3.11.9, selected for its compatibility with the required libraries. Python's rich ecosystem makes it an ideal choice for rapid development and machine learning integration.

- **Flask**
  Flask is the primary web framework used to manage the backend. It facilitates handling HTTP requests, routing, and interactions with Firebase. Flask's lightweight and modular architecture makes it suitable for building flexible and scalable applications.

- **FireBase**
  The Firebase Admin SDK is used for communication with Firebase services. It handles database interactions (Firestore), file storage (Firebase Storage), and user authentication, ensuring smooth integration with the backend.

- **OpenCV (cv2)**
  OpenCV is employed for image preprocessing tasks in the otoscope's image analysis. It performs operations such as resizing, filtering, and transformations, which are crucial for preparing images for deep learning models.

- **TensorFlow**
  TensorFlow powers the deep learning models integrated into the app for analyzing otoscope images and detecting ear infections. Using the Keras API, TensorFlow facilitates training, deploying, and running machine learning models for accurate predictions.

- **Uuid**
  This library generates unique identifiers for naming uploaded files, ensuring no file overwriting in Firebase Storage.

- **Os**
  The OS library facilitates interaction with the operating system for file handling, such as creating temporary files for image storage and management.

- **Tempfile**
  Tempfile is utilized to create temporary files for processing and uploading

images to Firebase Storage.

- **Kivy**
  Kivy is the main GUI framework used for the app's user interface. It provides tools for creating interactive, multi-touch applications with modern design elements. Kivy’s flexibility allows the AutoScope app to support both mobile and desktop environments seamlessly.

- **Requirements.txt**
  Essential part of the *AutoScope* app, listing all the dependencies and libraries required to run the software smoothly. It ensures that the environment setup is consistent across different systems, enabling developers and users to install the necessary packages with a single command. By including this file, the app facilitates easy deployment, compatibility, and maintainability, making it straightforward to replicate the working environment for testing, development, or production purposes.

3. Code Review

his section of the guide provides a comprehensive analysis of the codebase for the *AutoScope* app. It explains the functionality of each component, its purpose, and its role within the system architecture. Best practices for maintaining the code, ensuring scalability, and enabling future enhancements are also discussed. This review aims to equip developers with a thorough understanding of the backend's inner workings, ensuring they can manage and extend the system effectively.

3.1 Client Side

The client-side code of the *AutoScope* app is designed to provide a seamless user experience through a modular structure.

3.1.1 Icons Folder

The Icons folder in the client-side codebase is dedicated to storing visual assets used throughout the application. It contains icon images that enhance the user interface, making it more intuitive and visually appealing. These icons play a key role in navigation, button design, and representing various features or actions within the app. By centralizing these assets, the folder ensures consistent styling and simplifies the process of updating or adding new icons as needed.

3.1.2 Screens Folder

The screens folder serves as a crucial component of the client-side codebase, housing the .kv files for all the application's screens. These files define the layout, styling, and structure of each screen within the app, enabling a clear separation of design from

logic. By organizing the UI components in .kv files, the folder ensures a modular and maintainable approach to building the app's interface, allowing for easy updates and consistent design across the various screens.

3.1.3 config.py

The config.py file contains essential configuration settings for the application, including the SERVER_URL. In this case, SERVER_URL = "https://ndvp39.pythonanywhere.com" specifies the base URL for the backend server that the application interacts with. This centralized configuration allows the client-side code to easily communicate with the server for operations like data fetching, user authentication, and image processing. By storing the server URL in a separate configuration file, the app ensures maintainability and flexibility, enabling quick updates or changes to the backend URL without modifying the main application logic.

3.1.4 Screens Files

These .py files define the core logic and functionality for each screen in the application. These files handle user interactions, such as button clicks and input events, and ensure a seamless transition between different parts of the app. They coordinate with the backend to fetch or send data as required and manage the state of the application for each specific screen. By linking with their respective .kv layout files, these scripts ensure that the visual components on each screen respond dynamically to user actions or updates in the app’s data. They play a crucial role in maintaining the application's flow, ensuring that every feature operates cohesively and provides an intuitive user experience. The structure of these files also ensures modularity, making the app easier to maintain and extend.

3.1.5 main.py

The main.py file serves as the entry point for the *AutoScope* application, orchestrating the app's structure, screen navigation, and global configurations. It defines the *AutoScopeApp* class, which inherits from Kivy's App class and manages the lifecycle of the application. The app utilizes a *ScreenManager* to handle transitions and navigation between different screens, ensuring a smooth and intuitive flow.
go_back function is common function that is used in the screens to move back to previous screen.
The build method is responsible for constructing the main layout of the app, incorporating a breadcrumb widget for navigation context and dynamically loading .kv files corresponding to the app's screens and widgets.
The *on_login_success* method is a key feature of the application, dynamically adding screens to the *ScreenManager* after a successful login. This approach ensures that screens are only loaded when needed, optimizing memory usage and improving performance. Each screen is instantiated and assigned a unique name, allowing seamless transitions between them. The app also maintains a global dictionary, *user_details*, to store user-specific data accessible across all screens. Additionally, the use of a *SlideTransition* ensures visually appealing animations during screen changes.
The file incorporates modularity by separating the logic for each screen into individual .py files and their associated .kv files, maintaining a clean and maintainable codebase. The inclusion of custom widgets, such as the breadcrumb for navigation

and the *ClickableImage* class for interactive image elements, enhances the app's usability and interactivity.

3.1.6 __pychache__ folder

The __pycache__ folder is automatically generated by Python and is used to store compiled bytecode files (.pyc) for the app's Python scripts. These files are created when Python modules are imported to improve performance by allowing the interpreter to execute the precompiled bytecode instead of recompiling the source code each time the app runs.
This folder works automatically and should not be changed manually

3.1.7 OtoScope_Video_Screen.py

The *OtoScopeVideoScreen* class in the application manages and displays video streams from connected cameras. It uses OpenCV to detect available cameras via the *get_available_cameras* method and populates a dropdown menu using *update_camera_spinner*. Users can select a camera using *select_camera*, which updates the camera index and starts the video stream with *start_video_stream*. Video frames are captured and displayed on the screen using *update_video*, which converts the frames into Kivy Texture objects and assigns them to the *video_area* widget. The *capture_image* method allows saving the current video frame as an image and transitioning to the *ChosenImageScreen*. Cleanup operations, such as releasing the camera resource, are handled in on_stop and *go_back*. Additionally, *on_pre_enter* and *on_enter* manage UI updates, like resetting the dropdown and clearing previous textures when the screen is entered.

3.1.8 Widgets Folder

several custom Kivy components designed for enhanced UI customization and functionality. These include a *Breadcrumb* widget for navigation paths, featuring dynamic label and arrow updates; a *RoundedTextInput* with customizable background color and rounded corners; multiple *RoundedButton* variants, including a button with icon support and press/release color change effects; a *RoundedCostumButton* with dynamic color and rounded rectangle styling; and a *FeedbackPopup* for displaying messages with an optional callback on dismissal. Each class incorporates dynamic resizing and styling through bindings and the Kivy canvas, enabling adaptability and consistency across the UI. These components enhance the application's visual appeal and user experience while maintaining a modular design for future adjustments.

3.2 Server Side

The server-side code of the *AutoScope* app is designed to ensure efficient processing, secure data management, seamless communication with the client, and effective integration with the database and trained model for storing data and performing accurate analysis.

3.2.1 Requirments.txt

The requirements.txt file lists all the necessary Python dependencies for the backend environment. It specifies the packages and their respective versions required to run the application, ensuring consistent and reproducible setups across different environments.
By running the command *pip install -r requirements.txt*, all the listed dependencies will be installed, helping the server-side components function smoothly.

3.2.2 Trained Model (.tflite)

The .tflite files in the server-side of the ear infection detection project contain the trained TensorFlow Lite model, which is optimized for efficient inference on mobile and edge devices. That model is initially trained in Google Colab, where the TensorFlow models is fine-tuned and then converted to the .tflite format to ensure they are lightweight and optimized for performance. Once the models is trained and converted, it is deployed to the server, where it's used to process and analyze images captured. The server utilizes that .tflite model to generate accurate diagnoses based on the captured image, determining whether an ear infection is present.

3.2.3 app.py

The provided app.py file is a Flask-based web server used in the *AutoScope* project. The app interacts with various external services, such as *Firebase* for authentication, *Firestore* for storing user data, *Firebase* Storage for image storage, and TensorFlow for model inference. It provides endpoints to handle several core operations, including user sign-up, login, updating user details, saving diagnostic results, and analyzing images with a trained *TensorFlow Lite* model. It also integrates email functionality for communication purposes.
The app uses environment variables to store sensitive data, such as Firebase credentials and Gmail credentials, which are loaded using the *dotenv* package.

3.2.3.1 Home Route (/test)

This route is a simple health check endpoint for the server. It is accessed through a *GET* request, and when invoked, it sends back a basic response with the message *AutoScope* Server is running." It helps ensure that the server is up and running, especially useful for debugging or when integrating the backend with other services.

3.2.3.2 Save Result (/api/save_result)

This function handles saving diagnostic results along with an image. It accepts a *POST* request that includes the diagnostic result and the image. The image is uploaded to Firebase Storage, where it's saved under a unique name generated by combining the current timestamp with the user’s ID. After the upload, the image URL is retrieved from *Firebase* Storage and is then stored in *Firestore* under the user’s document. This also involves updating the document with the diagnostic result and the timestamp. If there’s an issue at any point, such as with the image upload or *Firestore* update, it returns an error message with details about what went wrong.

3.2.3.3 Analyze Image (/api/analyze_image)

This endpoint processes an uploaded image, prepares it for analysis, and runs it through a *TensorFlow Lite* model to classify the image. Upon receiving a POST request containing the image, the image is resized and converted into a format suitable for model input. The *TensorFlow Lite model* performs image classification, and the function returns the predicted class (like ‘aom’, ‘com’, or ‘normal’) along with the model's confidence score in that prediction. This function helps in analyzing ear images for infection detection.

3.2.3.4 Sign Up (/api/signup)

This function allows a new user to sign up for the *AutoScope* app by providing their full name, email, password, phone number, and gender. The function first registers the user in Firebase Authentication, where the email and password are used to create the user account. After successful authentication, the user’s additional details (like name, phone number, and gender) are saved in the *Firestore* under the 'Users' collection. If the registration process is successful, the function returns the UID of the newly created user, which can be used for future references or user-specific actions.

3.2.3.5 Login (/api/login)

The login function authenticates users by verifying their credentials (email and password). A POST request with the email and password is sent to Firebase Authentication. If the credentials are correct, the function retrieves the user’s details (like name, email, phone number, gender) from *Firestore* and sends them back in the response. If there’s an issue with the login (e.g., incorrect credentials or an authentication error), an error message is returned to indicate the failure.

3.2.3.6 Save Settings (/api/save_settings)

This endpoint is used to update a user’s profile settings, such as name, email, phone number, and gender. It processes a POST request containing the updated details and first updates the *Firebase Authentication* profile with the new email and other information. Then, it updates the corresponding *Firestore* document in the 'Users' collection with the updated data. If the update is successful, it sends a success message back, otherwise, it returns an error message with the details of what went wrong.

3.2.3.7 Get History (/api/get_history)

This function retrieves a user's diagnostic result history from *Firestore*. When the user makes a request, it queries the *Firestore* database to fetch all the documents associated with the user’s UID in the 'Results' collection. Each document contains details about past analyses, including the diagnosis, image URL, and timestamp. The results are returned as a list of dictionaries, allowing the user to view their previous diagnostic results.

3.2.3.8 Send Email (/api/send_email)

This function allows the app to send an email using the Gmail SMTP server. It accepts parameters for the email's subject, content, and recipient. The function uses the SMTP protocol to send the email by connecting to Gmail's mail server with authentication credentials stored as environment variables. If the email is successfully sent, it returns a success message; otherwise, an error message is returned indicating the failure reason, such as issues with the email configuration or SMTP connection.

3.3 Model Training

The deep learning model in this application uses TensorFlow and ResNet50 to classify otoscope images into "Infected" or "Normal." Regular maintenance of the model ensures its accuracy, reliability, and adaptability to new datasets or use cases. This section outlines the functions involved in the model pipeline and provides guidance on updating or troubleshooting them when necessary.

3.3.1 Dataset Fetching & Seperation (fetch_dataset_from_drive, split_dataset)

To update the dataset or fetch a new version, replace the file ID or shared link in this function. Ensure that the Drive permissions allow access, and confirm that the dataset is downloaded to the correct location specified in the function.
split_dataset organizes the downloaded dataset into training, validation, and testing subsets. It divides the data into appropriate proportions and saves the subsets in separate directories. If you need to adjust the split ratio, modify the parameters in this function. Additionally, ensure that the dataset is balanced across the classes to avoid bias. For any changes, verify that the output directories contain the correct proportions and that all files are accessible for further processing.

3.3.2 Data Preprocessing (preprocess_images)

This function prepares images for the model by resizing them to 224x224 pixels and applying the *preprocess_input* method from ResNet50. It also supports data augmentation for training images to improve generalization. To modify the preprocessing steps (e.g., change image dimensions or include additional augmentations), edit this function in the preprocessing script. Ensure changes are consistent with the model's input requirements.

3.3.3 Model Training (train_model)

This function handles the training process, including loading the dataset, defining the model architecture, and applying callbacks such as early stopping and learning rate scheduling. If you need to retrain the model on a new dataset or adjust hyperparameters (e.g., learning rate, batch size), update the relevant arguments in this function. Ensure the new dataset is properly preprocessed using *preprocess_images.*

3.3.4 Model Evaluation (evaluate_model)

The evaluation function computes metrics like accuracy, confusion matrix, and ROC curves. If additional metrics are required, modify this function to include them. For troubleshooting performance issues, review the outputs of this function to identify potential bottlenecks, such as data imbalance or overfitting.

3.3.5 Saving and Loading the Model (save_model, load_model)

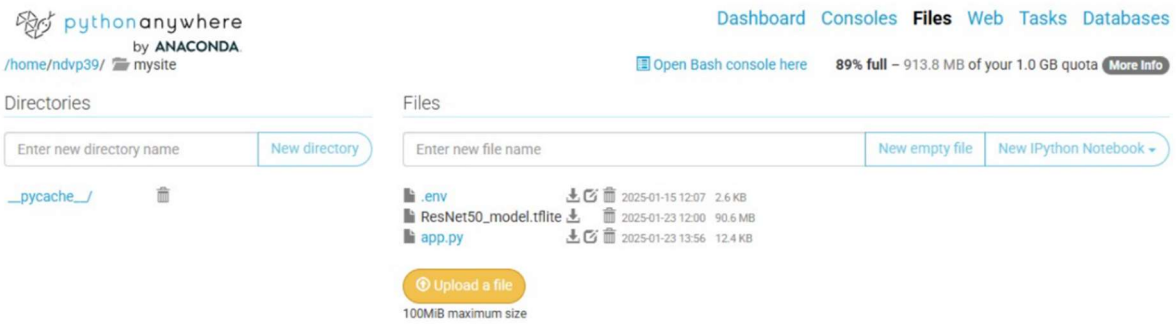
The *save_model* function saves the trained model and its weights, while *load_model* reloads them for inference. To change the model's save path or format (.h5 or TensorFlow SavedModel), update these functions accordingly.

3.3.6 Training History (save_training_history, load_training_history)

These functions manage the JSON file that records the model's training history. This file is useful for analyzing training trends or reproducing the training process. If you encounter errors in loading the history file, verify its structure and ensure it is properly serialized during saving.

4. Deployment

The backend of the *AutoScope* application is deployed on **PythonAnywhere**, a cloud-based platform that allows running Python applications. The deployment process involves setting up the necessary files, configuring the environment, and ensuring the Flask application is properly executed via WSGI.
To begin with, all backend functionalities, including user authentication, history retrieval, and AI model integration, are contained within a single file, app.py. The trained AI model, stored as *ResNet50*_model.tflite, is also uploaded to the project directory to enable real-time image analysis. Additionally, an .env file is included to manage environment variables, ensuring secure access to sensitive information such as API keys and database credentials.



setting up environment variables, including PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION, which ensures compatibility with TensorFlow Lite. The script then appends the Python 3.9 site-packages directory to sys.path, ensuring all necessary dependencies are accessible. It

also includes the project directory (/home/ndvp39/mysite) in the system path to make the application files discoverable. Finally, the Flask app from app.py is imported and assigned to application, which is a requirement for PythonAnywhere's WSGI interface.

/var/www/ndvp39_pythonanywhere_com_wsgi.py

```python
import sys
import os
from dotenv import load_dotenv

os.environ["PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION"] = "python"


# Add the path to Python 3.9 packages
sys.path.append('/home/ndvp39/.local/lib/python3.9/site-packages')

project_folder = os.path.expanduser('/home/ndvp39/mysite')  # adjust as appropriate
load_dotenv(os.path.join(project_folder, '.env'))

# add your project directory to the sys.path
project_home = '/home/ndvp39/mysite'
if project_home not in sys.path:
    sys.path = [project_home] + sys.path

# import flask app but need to call it "application" for WSGI to work
from app import app as application  # noqa

```

Once the necessary configurations are in place, the deployment process involves a few key steps. First, all required files, including app.py, the AI model, and the .env file, are uploaded to PythonAnywhere. The wsgi.py file is then modified if needed, ensuring all paths and dependencies are correctly set. After making these adjustments, the web application is restarted from the PythonAnywhere dashboard to apply the changes. If any issues arise, logs stored in /var/www can be checked for debugging. By following this deployment process, the backend is successfully hosted on PythonAnywhere, making it accessible for handling API requests and processing image analysis through the AI model.